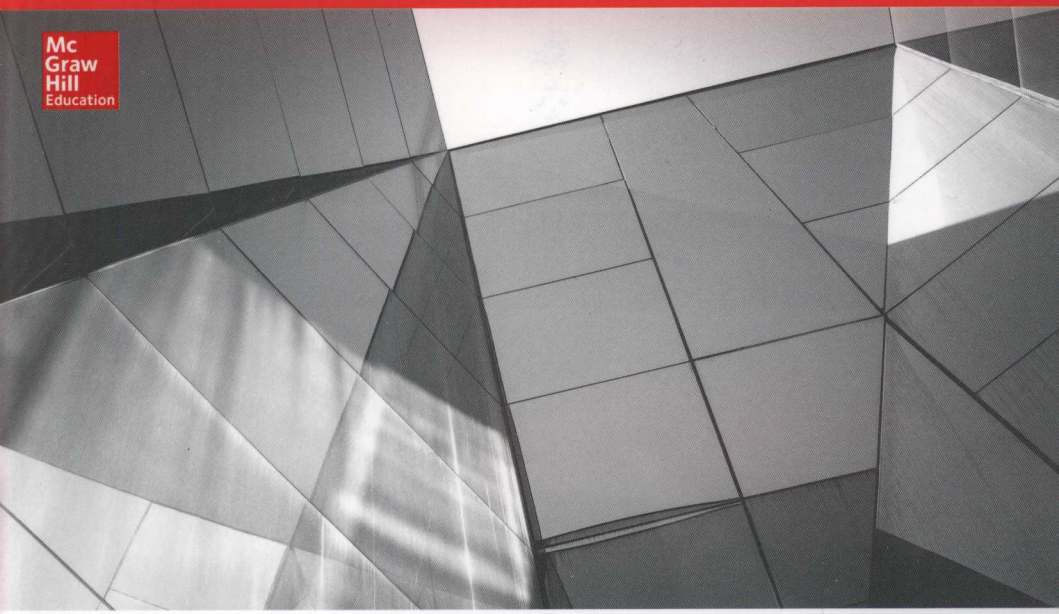
ORACLE



Java WebSocket Programming

Java WebSocket编程

开发、部署和保护动态Web应用

[美] Danny Coward 著
刘建 夏先波 译

清华大学出版社

Oracle 官方系列丛书

Oracle数据库

Oracle Database 11gR2性能调整与优化
OCP/OCA认证考试指南全册：Oracle Database 11g(1Z0-051, 1Z0-052, 1Z0-053)
Oracle Solaris 11系统管理完全参考手册
Oracle Database 11g初学者指南
Oracle Database 11g SQL开发指南
Oracle Fusion Middleware 11g架构与管理
Oracle Database 11g RAC手册(第2版)
Oracle Data Guard 11g完全参考手册
Oracle安全实战——开发安全的数据库与中间件环境
Oracle Database 11g RMAN 备份与恢复
Oracle Database 11g R2高可用性：
　　使用Grid Infrastructure、RAC和Data Guard最大限度提高可用性(第2版)
Oracle Database 10g SQL开发指南
基于Linux平台的Oracle Database 10g管理
Oracle WebLogic Server开发权威指南
OCA认证考试指南(1Z0-047)：Oracle Database SQL Expert
OCA 认证考试指南(1Z0-051)：Oracle Database 11g SQL Fundamentals I
OCA 认证考试指南(1Z0-052)：Oracle Database 11g Administration I
OCP认证考试指南 (1Z0-053)：Oracle Database 11g Administration II
OCA Java SE 7 Programmer I 认证学习指南(Exam 1Z0-803)

Java WebSocket 编程

开发、部署和保护动态 Web 应用

[美] Danny Coward 著

刘 建 夏先波 译

清华大学出版社

北 京

Danny Coward
Java WebSocket Programming
EISBN：978-0-07-182719-5

图书在版编目(CIP)数据

Java WebSocket 编程　开发、部署和保护动态 Web 应用 /(美)　科沃德(Coward, D.) 著；刘建，夏先波 译. —北京：清华大学出版社，2015
书名原文：Java WebSocket Programming
ISBN 978-7-302-40807-9

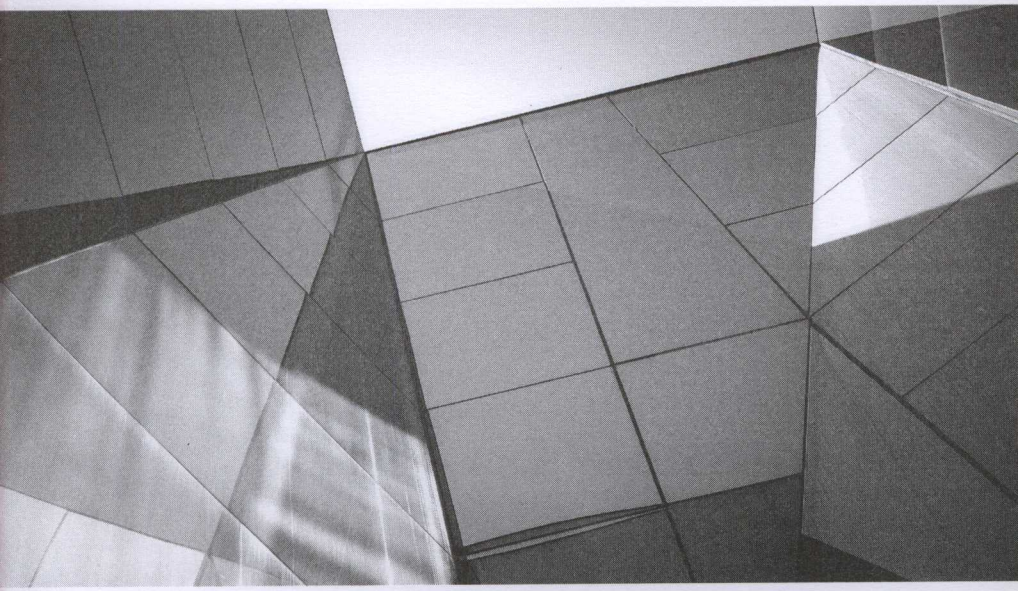
Ⅰ. ①J… Ⅱ. ①科… ②刘… ③夏… Ⅲ. ①JAVA 语言—程序设计 Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字(2015)第 162976 号

责任编辑：王　军　于　平
装帧设计：孔祥峰
责任校对：邱晓玉
责任印制：刘海龙

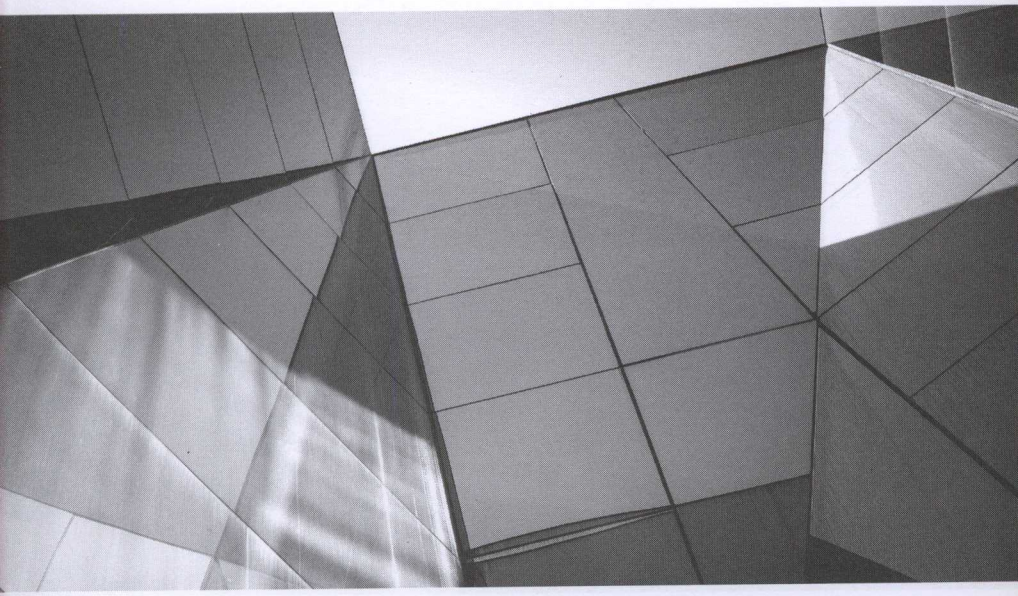
出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：清华大学印刷厂
经　　销：全国新华书店
开　　本：148mm×210mm　　**印　　张**：8.25　　**字　　数**：200 千字
版　　次：2015 年 8 月第 1 版　　**印　　次**：2015 年 8 月第 1 次印刷
印　　数：1～3500
定　　价：39.00 元

产品编号：057242-01



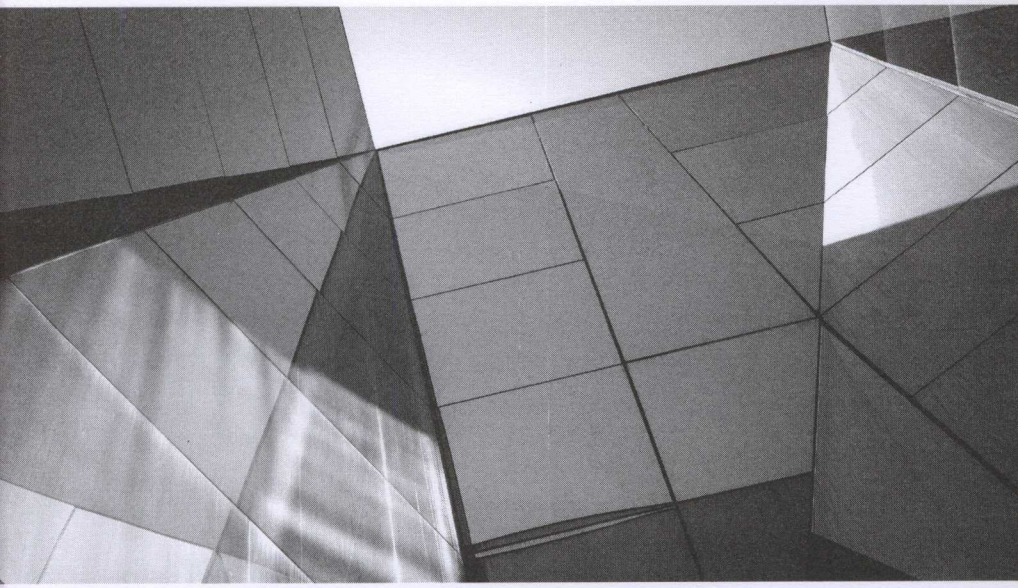
作者简介

Danny Coward是Oracle的首席架构师和Web架构师。他是Java EE、Java SE/JavaFX中WebSocket Java API的规范领导者。Coward在Oracle主导WebSocket工作，他是Java WebSocket编程方面的权威专家。Coward在Java软件的所有方面——从Java ME到Java EE再到Java FX技术的建立——都具有丰富的专业经验。



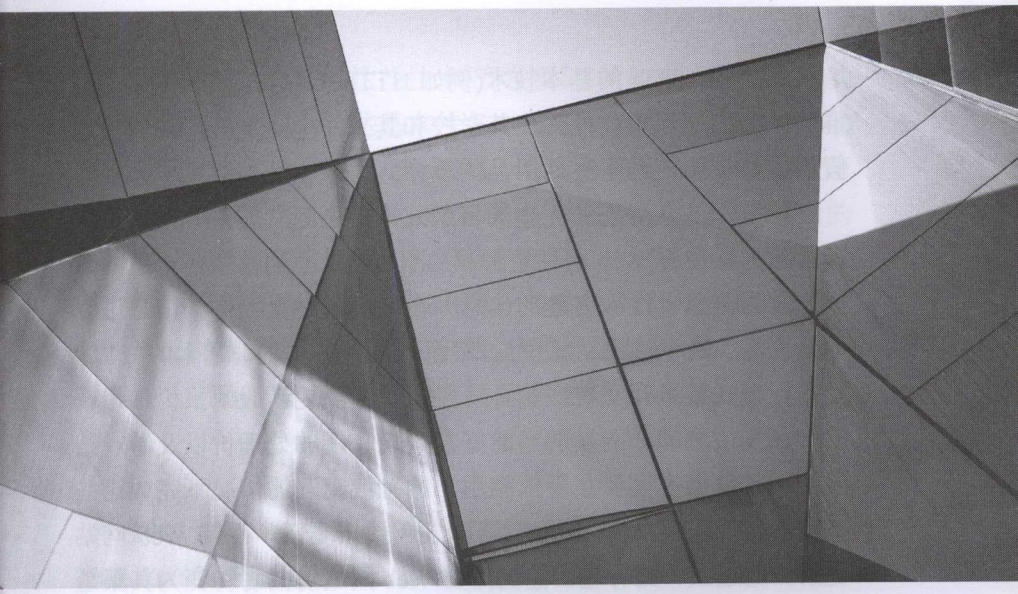
技术编辑简介

Santiago Pericas-Geertsen博士是Oracle公司Sun Glassfish机构的主要技术人员，也是Avatar项目的架构师和技术主管。Santiago是JSR 339、JAX-RS 2.0的规范领导者。在Sun Microsystems公司就职时，Santiago是Glassfish Mobility Platform的技术主管，是Fast Web Services项目的开发者和领导者，也是World Wide Web Consortium(W3C)计划中的参与者和编辑。他持有两项美国专利：7647415和7716577。Santiago的博客在Java.net上，推持号是@spericas，并且他还出席了多场面向学术和产业的会议。



致　谢

非常感谢 Santiago，他在本书写作期间给予了深思熟虑的评论；还要感谢 McGraw-Hill Education 的 Brandi Shailer 和 Amanda Russell，他们促使我保证了写作的进度。



前　言

对于 Web 开发人员迅速发展的工具箱而言，WebSocket 协议是一种新的网络协议。它除了作为 HTML 5 的核心技术外，还被从桌面到平板电脑和智能手机的所有主流浏览器迅速采用，不过为什么 Web 开发人员还要关心另一种网络技术？

长轮询

在 2000 年之前，全世界的多数主流公司都使用 Web。在发展中国家，个人电脑的革命使得大部分家庭都至少通过一个通道接入互联网。企业迅速地在互联网上建立其 Web 站点，作为一种展示产品和服务的手段，并且作为交付这些产品和服务的一个日益

增长的渠道。Web 的基本技术(例如 HTTP、HTML 和 JavaScript)推动了人们如何与其他人、其学校和其工作地点交互，如何规划假期，甚至如何购买生活用品的革命。

Web 网站从静态和无趣的目录式风格不断发展，开发人员找到了新方法使得 Web 网站更加具有交互性。他们指望在合适的时候通过为浏览者注入有趣的信息并在必要时对页面中的信息进行更新来为 Web 网站添加活力。然而开发人员发现，基本的 HTTP 及其标记式技术存在着限制。开发人员需要更新股票报价、最新出价、登录到同一网站的当前好友列表、新的处理价以及游戏结果。同时他们也需要在不依赖用户的持续交互的情况下完成这些事情。他们需要从 Web 服务器端发起数据更新，保持 Web 网站更新、更迷人、更有趣。他们需要 Web 网站访问者转变为其渠道的一个观察者，并且他们需要访问者为了获取推送给他们的信息做尽可能少的事情。

在之后的几年中，开发人员通过各种非正规手段来完成更新一个网站最新的各种类型的信息到所有当前访问者的任务。最明显的手段是通过浏览器向服务器轮询更新。开发人员将一小段 JavaScript 嵌入相关的 Web 页面中，强迫浏览器以预定的间隔周期刷新整个页面。无论获取的数据是否需要刷新，此方法都将刷新所有的数据。即使除去获取非必需数据之外，此方法的网络延迟也是比较明显的，所以其用户体验比较糟糕。

稍微复杂一些的处理方式是使用 HTTP Keep Alive 机制。在这种机制中，Web 页面中的 JavaScript 代码将保持打开一个长生命周期的 HTTP 连接，如同一个持续不断的软件下载，它将定期地使用新信息进行更新。关于浏览器和服务器应保持连接打开多久方面的巨大差异导致了大量问题。一般情况下，客户端的浏览器将需要频繁地重新打开连接，而无论其是否从服务器获取数据。

随着开发人员接触这些技术，开发框架(例如 Comet 和 AJAX)逐步开始支持并包含这些基本技术。在某种程度上，它们可以隐藏这些基本技术的一些不足。然而，有两个基础问题即使是最好的实现也不能克服。首先，HTTP 是一个发送简单信息的昂贵的网络协议。仅仅请求简单的股票报价更新，连接上下文在每次请求中都会被重新调用：例如限定客户端和服务器平台的所有头信息、认证属性、负载描述等。其次，更糟糕的是，无论服务器是否有新的信息传送，都需要建立昂贵的连接。

WebSocket 的引入

2009 年，WebSocket 的引入工作始于一项允许客户端与服务器建立一个轻量级的连接，并允许双向通信和一个轻量级的内容模型的技术。服务器能够仅在必要时将数据推送到已连接的客户端。一旦连接建立后，在每次发送消息时无须重新创建连接上下文，客户端和服务器都将有办法发送简单信息。

不必要的更新的日子即将结束。

为了理解轮询方式如何浪费网络资源，考虑一个拍卖网站。物品可以发布在网站上，在定义的时间段内，用户可以对物品进行投标，在投标时间结束后，物品被卖给出价最高的用户。在拍卖的整个过程中，任意访问投标页面的用户都能够看到当前的出价并使用此信息来决定更高的出价。若网站提供一个极其令人满意的物品来拍卖(一个稍微使用过的 iPod，毕竟它是 2003 年的产品)，同时此物品的拍卖时间仅仅只有一个小时。假设当这个 iPod 有新的出价时，网站需要仅仅传送一个简短消息，包含新的价格以及可能的一些附加信息(例如，竞标者的线上名称)。我们估计这些信息总是符合 64 个字节。若考虑到所有已登录的用户，为了获取新的信息，一些 Cookie 信息需要在 HTTP 请求中传送。连同

内容类型头、可能两三个特定于应用的头、内容长度、浏览器 ID 等一起，我们估计头信息大约为 512 字节。现在假设平均有 100 个用户登录到此网站，在拍卖过程中平均每 30 秒有一个新的出价。假设出价的频率并不是平均分布的；在某一时刻，出价也许相隔几秒钟(接近于拍卖时间的结束)，而在平时(拍卖时间开始)出价可能相隔几分钟。竞标者将希望最近价格信息一直可用，所以他将谨慎地每两秒钟刷新一次价格；否则，因为在竞标者获取更新前另一个出价已完成，当出价不被通过时，竞标者会变得失意。下面将为了获取更新发送的所有数据进行合计：

平均每分钟 30 次更新，总计 60 分钟 = 1800 次更新
每次更新带来大约 512 字节的头信息
发送和接收的头信息总计为：
1800 × 512 字节 = 921 600 字节 = 900KB

现在，若每次更新包含 64 字节。同时在此拍卖中，假设有 120 次出价。因此所有更新信息为：120 × 64 字节 = 7.5KB。

因此，有用数据与重复数据的粗略的效率比计算是 7.5 / 900 = 0.8%。

这并不是一个理想的分数，并且在更长时间的拍卖下，我们甚至不知道此效率比是如何降低的。

WebSocket 旨在通过仅在建立连接时发送上下文信息，并且在连接建立后允许连接的两端在携带着极少的标识信息的上下文信息的情况下发送消息(即使是同时发送)来显著增加这种类型的网络效率。

这样，连接到 Web 服务器的 Web 页面能够仅在服务器决定它们需要更新时才接收更新。同时当发送此更新时，它不需要使用关于连接的大负载的上下文信息来使得消息的负荷太重。

WebSocket 协议介绍

WebSocket 协议是一个网络协议，允许两个相连的端在一个单一 TCP 连接上进行全双工消息通信。对 WebSocket 的一个合适的类比是打电话。当打电话时，你通过拨号初始化一个电话呼叫。如果你尝试呼叫的人通过拿起话筒接受了呼叫，连接就已建立。当连接活跃时，如果乐意，双方可以同时说话(虽然并不推荐此种自由交谈)，同时即使在说话时双方也都可以听到正在说些什么。这就是全双工通信的含义所在。不管是否有人在讲话，连接一直保持活跃，直到双方中的一方决定挂断电话。

在 WebSocket 的场景中，连接通过 HTTP 和 WebSocket 端点交互的方式建立。连接的发起者发送一个专门制定的 HTTP 请求，其中包含其希望连接的 WebSocket 端点的 URL。它开始活动并被称为打开阶段握手。如果服务器愿意接受连接，服务器制定一个称为打开阶段握手响应的特殊的 HTTP 响应并将其发送回客户端。此时，TCP 连接可能已经建立，能够保证 WebSocket 消息的往返传递。连接将一直保持活跃直到任意一方决定终止连接，或者是某些外部因素导致连接关闭(例如，非活跃期太长导致超时或者物理网络的问题)。

尽管没有任何特定的协议需要部署设置，WebSocket 主要用来作为托管在 Web 服务器上的 Web 应用和浏览器客户端之间的通信机制。同样，WebSocket 连接可以在网络间的任意两端建立，而不必是浏览器和 Web 服务器上。然而，由于技术的起源，WebSocket 技术最直接的机会是为静态 Web 站点带来活力，并且能够很容易地通过实时数据和活动来增强 Web 网站和 Web 应用。

在设置中，存在于浏览器中的 WebSocket 是使用已经被 W3C 标准化的称为 JavaScript WebSocket API 的一个 JavaScript API 创

建的。存在于托管在 Web 网站的 Web 服务器上的 WebSocket 可以通过多种语言来开发。WebSocket 的普及如此突然，以至于多种不同语言和 Web 平台都开始支持它。特别是，Java 平台已经快速建立了对 WebSocket 的支持。本书的主要关注点是探索这个新的 Java API 的基础设施，尽管一些示例将依赖于那些使用 JavaScript API 来支持 WebSocket 的代码。

Java Web Socket API是最近的Java EE 7平台的核心特性，并且任何已经熟悉用其他Java技术或者基于Java的技术来构建Web应用(例如Java Servlet、JSP、JavaServer Faces或者与这些技术相关的任何其他技术)的Web程序员应该熟悉Java平台中的WebSocket并且会考虑通过引入此技术为现有或者未来的应用带来新鲜的和现代的感觉。通过本书，你将学习如何编写WebSocket应用。还将学习Java WebSocket API的所有主要设施，包括WebSocket应用可用的配置选项和多种消息模式、可以在何处存储应用状态、如何配置WebSocket使得它仅可以被安全地访问，以及如何将WebSocket集成到Java EE应用中。你在本书中每一章所学习的知识都将通过示例应用来演示。

本书共包含 8 章内容。

第 1 章：Java WebSocket 基本原理

该章深入介绍了第一个 WebSocket 应用：Echo 应用。虽然比较简单，但是介绍了 Java WebSocket API 的主要特性，从而形成了本书中其他章节的基础。

第 2 章：Java WebSocket 生命周期

该章介绍了在 WebSocket 应用中创建的主要组件——WebSocket 端点——的生命周期。此生命周期定义了一个框架，通过此框架可以管理 WebSocket 端点使用的资源，并且最重要的

是定义了如何拦截 WebSocket 消息。此生命周期通过 Lifecycle 应用来举例说明。此示例为用户展示了一系列的交通信号灯，以便强调 WebSocket 端点生命期中的关键阶段。

第 3 章：消息通信基础

该章使用团体合作绘图应用作为其示例应用，介绍了Web应用中发送和接收消息的基本方面。此示例使用Java应用作为客户端，所以该章也同样展示了Java WebSocket API作为客户端的用法。

第 4 章：配置与 Session

该章阐述了 Java WebSocket API 中的两个最重要的对象：WebSocket 会话对象和端点配置对象。其中前者表示一个和 WebSocket 对等端的会话，后者保存端点的配置信息。在一个在线聊天的示例应用中，我们使用这些对象说明其特性。

第 5 章：高级消息处理

该章详细介绍了开发人员用于发送和接收 WebSocket 消息的所有可用选项。我们以第 3 章为基础介绍了一些高级话题，例如消息编码和解码策略、同步和异步消息模式。该章中的示例应用使用一个用户界面来阐明 API 中可用的消息选项。

第 6 章：WebSocket 路径映射

该章覆盖了路径映射的 9 条规则，讲解了 Java WebSocket API 中将 WebSocket 端点发布到一个 URI 使得对等节点能够连接上的所有可用选项。该章使用一个简单的股票投资组合示例说明精确路径匹配、模板映射、查询字符串等技术，并且讨论了你可能选择某种技术而非另一种技术的各种情况。

第 7 章：保护 WebSocket 服务器端点

该章介绍了如何将对 WebSocket 端点的访问限制为仅是 Web 应用的某种用户，还介绍了如何确保 WebSocket 端点间的通信保持为私有。我们回顾了股票投资组合应用，使用安全技术来保护此应用并且使之个性化。

第 8 章：Java EE 平台中的 WebSocket

该章首先介绍如何将 WebSocket 端点集成到大规模的 Java EE 应用中。该章改进了第 4 章中的聊天应用，从而利用 Java EE 平台中的两个关键组件：Java Servlet 和 EJB，提供了一些在 WebSocket 端点和其他 Java Web 组件之间，以及和企业 JavaBean 间共享应用信息的方法。

目标读者对象

本书适合于如下读者：

希望为 Web 应用增加交互特性的 Web 开发人员
希望和 WebSocket 服务器应用交互的富客户端应用开发人员
有兴趣为支持 HTML5 的浏览器开发应用的 Java EE 开发人员

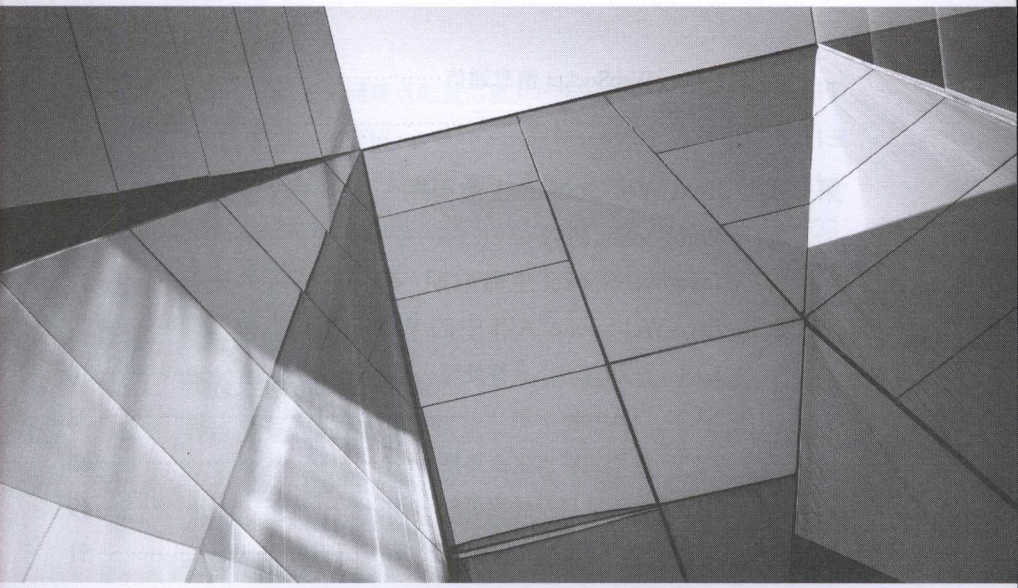
本书假设你具备一些 Java 编程语言的运用知识、一些 Java SE 平台的经验和一些 JavaScript 编程语言的知识。此外，尽管不是必要的，但具备一些 Web 应用的开发经验可能是有用的。

获取示例应用

所有的示例应用代码都可以从 Oracle Press 网站 www.OraclePressBooks.com下载。文件包含在一个压缩文件中。一旦下载了此压缩文件，就需要解压其内容。示例运行于Glassfish 4.0应

用服务器上。Glassfish 4.0可以从http://glassfish.java.net/免费下载，并且可以使用NetBeans IDE进行构建。NetBeans IDE可以从http://netbeans.org/免费下载。

我希望本书能够激发你使用前所未有的想法来编写应用！



目　录

第 1 章　Java WebSocket 基本原理……1

1.1　创建第一个 WebSocket 应用……2

1.1.1　创建 WebSocket 端点……3

1.1.2　部署端点……5

1.1.3　创建 WebSocket 客户端……6

1.2　WebSocket 端点……9

1.3　编程式端点……10

1.4　深入 Echo 示例……17

1.4.1　部署阶段……17

1.4.2　接收第一个连接……19

1.4.3 WebSocket 消息通信……22
1.5 本章小结……23

第 2 章 Java WebSocket 生命周期……25
2.1 WebSocket 协议……26
2.2 Java WebSocket 生命周期……27
2.3 Java WebSocket API 中的 WebSocket 生命周期……29
2.3.1 注解式端点事件处理……30
2.3.2 Lifecycle 示例……38
2.3.3 编程式端点生命周期……46
2.3.4 实例数目及线程机制……49
2.4 本章小结……51

第 3 章 消息通信基础……53
3.1 消息通信概述……54
3.1.1 发送消息……54
3.1.2 接收 WebSocket 消息……61
3.2 DrawingBoard 应用……73
3.3 消息通信和线程……94
3.3.1 WebSocket 端点线程和消息通信……94
3.3.2 线程与编码器和解码器的生命周期……95
3.4 本章小结……95

第 4 章 配置与 Session……97
4.1 Session 状态和逻辑端点状态……98
4.2 Chat 示例……99
4.3 配置端点：ClientEndpointConfig 和 ServerEndpointConfig……111

4.3.1 提供和访问端点配置信息 …… 111
4.3.2 配置选项介绍 …… 113
4.3.3 WebSocket 子协议和 WebSocket 扩展 …… 115
4.4 WebSocket Session …… 123
4.5 本章小结 …… 128

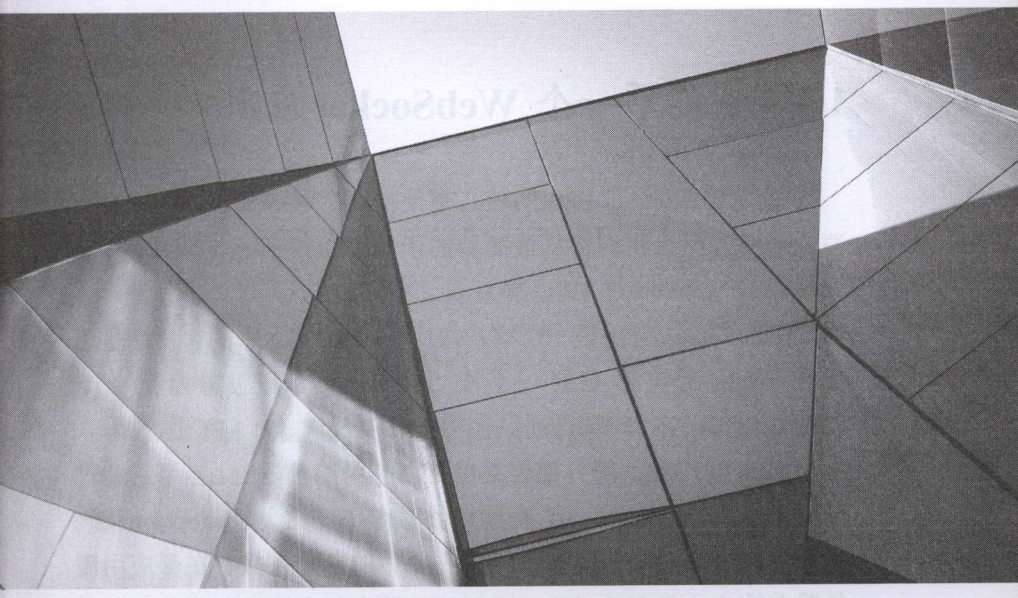
第 5 章 高级消息处理 …… 129
5.1 检查连接：Ping 和 Pong …… 130
5.2 异步发送 WebSocket 消息 …… 132
5.2.1 通过 Future 发送 WebSocket 消息 …… 133
5.2.2 通过 Handler 发送 WebSocket 消息 …… 135
5.2.3 何时通过 Future 发送以及何时通过 Handler 发送 …… 136
5.2.4 异步发送超时 …… 137
5.3 消息批处理 …… 138
5.4 缓冲、消息分片和数据帧 …… 140
5.5 保证消息传递 …… 142
5.6 发送消息 API 总结 …… 143
5.7 MessageModes 应用 …… 144
5.7.1 MessageModes 应用概述 …… 145
5.7.2 查看 MessageModes 应用的代码 …… 147
5.7.3 MessageModes 应用中需要注意的事情 …… 155
5.8 本章小结 …… 158

第 6 章 WebSocket 路径映射 …… 159
6.1 URI 术语 …… 160
6.2 WebSocket 路径映射 …… 161
6.2.1 精确 URI 映射 …… 161
6.2.2 URI 模板路径 …… 164

6.2.3 URI 模板匹配相关的 API……167
6.2.4 在运行时访问路径信息……173
6.2.5 查询字符串和请求参数……174
6.2.6 匹配优先级……177
6.3 Portfolio 应用……180
6.4 查询字符串、路径参数与 WebSocket 消息……186
6.5 WebSocket 路径映射 API 总结……188
6.6 本章小结……189

第 7 章 保护 WebSocket 服务器端点……191
7.1 安全的概念……192
7.2 Java WebSocket API 安全……193
7.2.1 认证……194
7.2.2 授权……199
7.2.3 私有通信……204
7.2.4 Java WebSocket 安全 API……207
7.3 Stock Account 应用……209
7.4 本章小结……217

第 8 章 Java EE 平台中的 WebSocket……219
8.1 Java EE 平台中 Java WebSocket 的角色……220
8.2 共享 Web 应用状态……222
8.2.1 HttpSession 与 WebSocket Session 的关联……224
8.2.2 HttpSession 示例……225
8.3 WebSocket 端点使用 EJB……230
8.4 新版 Chat 示例……235
8.5 本章小结……240



第1章

Java WebSocket 基本原理

本章介绍 Java WebSocket API 并粗略了解其功能。本章将深入介绍一个示例应用，此示例应用的服务器端简单地回显客户端发给它的任意消息。我们将通过此示例来阐述 Java WebSocket API 的主要特性。这样做，本章将建立本书其他部分所需要描述的主要特性的基础。如果你需要复习 WebSocket 协议的主要概念，请在继续学习之前先看看“前言”部分。

1.1 创建第一个 WebSocket 应用

既然 WebSocket 协议的核心是使得两个对等节点间能够进行消息通信，那么我们将从最简单的示例(EchoServer 应用)来开始介绍 Java WebSocket API。

EchoServer 应用是一个客户端/服务器应用。它的客户端是运行在 Web 浏览器上的一小段 JavaScript，其服务器是一个 WebSocket 端点。此端点通过使用 Java WebSocket API 进行编写，作为 Web 应用的一部分部署并运行在应用服务器上。当用户通过浏览器载入 Web 页面时，在用户的命令下，这个 JavaScript 代码片段会被执行。此代码执行时首先要做的事是连接存在于应用服务器上的 Java EchoServer WebSocket 端点并在连接之后立刻发送一条消息。运行在应用服务器上的 Java EchoServer 端点等待入站的连接，并且在其接收到一个已连接的客户端发送的消息时，立刻对其进行回复，确认已收到消息。

通过此应用，也就是 EchoServer，可以阐述已经创建的 WebSocket 端点的关键方面。虽然该示例特别简单，但是充分理解其创建和部署过程将有助于我们迅速理解 Java WebSocket API 的关键特性。在本示例中，我们不会详述任何关键特性，而是会给出一个全局图，可以将该示例作为学习 Java WebSocket API 的一个很好的起点，在本书的后续章节将会更详细地阐述这些特性。

此示例大体上包括两大部分：第一部分同时也是最重要的是开发并部署在服务器上的 Java EchoServer 端点；第二部分是运行于浏览器中的 Web 页面，它包含 JavaScript WebSocket 客户端。图 1-1 展示了该示例的部署总图。

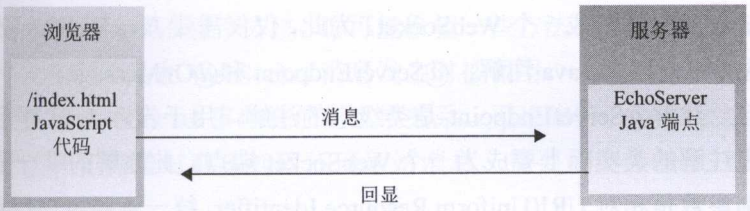


图 1-1　Echo 示例应用

1.1.1　创建 WebSocket 端点

幸运的是，创建第一个 WebSocket 端点并部署到应用服务器中非常简单。你所要做的全部事情是开发一个 Java 类，实现想让此端点所拥有的功能，并且了解怎样将 Java WebSocket API 中定义的少数几个 Java 注解应用于开发的 Java 类上。对于示例 EchoServer 来说，我们想要其实现从一个已连接客户端接收消息并立即将这个消息返回给此客户端的功能。

让我们立刻开始实现该功能。首先创建一个如代码清单1-1所示的名为 EchoServer 的 Java 类，然后为其添加一个方法，此方法的参数是 String 类型的，用于保存客户端发送的任意入站消息。此方法的返回值也是 String 类型。无论客户端何时发送一个入站消息，我们都希望此返回值可以用来将此消息发送回客户端。

代码清单 1-1：简单的 Java 对象(POJO)

```
public class EchoServer {
  public String echo(String incomingMessage) {
    return "I got this (" + incomingMessage + ")"
           + " so I am sending it back !";
  }
}
```

此时，你也许乐于知道你已经实现了此应用所需要的所有逻辑。所剩下的工作就是使用 Java WebSocket API 将此简单传统的

Java 对象转换成一个 WebSocket。为此，仅仅需要 Java WebSocket API 中的两个 Java 注解：@ServerEndpoint 和@OnMessage。

注解@ServerEndpoint 是类级别的注解，用于告诉 Java 平台它注解的类实际上要成为一个 WebSocket 端点。此注解的唯一强制参数是相对 URI(Uniform Resource Identifier，统一资源标识符)，开发人员希望这个端点在此相对 URL 之下可用。这个场景有点类似于给别人电话号码，使得大家可以通过电话号码打电话。为了使示例更加简单，将使用 URI /echo 来发布示例创建的新端点。

因此，对 EchoServer 类添加如代码清单 1-2 所示的注解：

代码清单 1-2：演化为 WebSocket 端点的 POJO 类

```
import javax.websocket.server.ServerEndpoint;

@ServerEndpoint("/echo")
public class EchoServer {
  public String echo(String incomingMessage) {
    return "I got this (" + incomingMessage + ")"
                + " so I am sending it back !";
  }
}
```

注意属性的名称是 value，它定义了端点将被发布在其下的路径，这意味着甚至不必将属性名称写到注解定义中。

现在代码已经包含了 WebSocket 实现能够运行的足够的信息，它将知道发布端点的 URI 空间。你可能想知道相对 URI 到底是相对于什么 URI：它相对于包含示例 EchoServer 端点的 Web 应用的上下文根。然而，因为并未告诉 WebSocket 实现当端点接收到消息时将会调用哪些方法，所以将 Java 类转换成 WebSocket 端点的过程还未结束。因为 EchoServer 类只有一个方法，在此示例中这点也许非常明显。然而，在更复杂的示例中，实现 WebSocket

端点的 Java 类有多个方法，你可能希望一些方法在 WebSocket 事件发生而不是在 WebSocket 消息发生时被调用，一些方法不需要直接和 WebSocket 事件有任何直接关系。无论如何，为了将实现方法标记为随时准备处理任何入站消息，需要使用方法级注解 @OnMessage。

不管你相信与否，你刚刚已经编写完了如代码清单 1-3 所示的第一个 WebSocket 端点。

代码清单 1-3：WebSocket 端点

```
import javax.websocket.server.ServerEndpoint;
import javax.websocket.OnMessage;

@ServerEndpoint("/echo")
public class EchoServer {

  @OnMessage
  public String echo(String incomingMessage) {
    return "I got this (" + incomingMessage + ")"
            + " so I am sending it back !";
  }
}
```

1.1.2　部署端点

部署 EchoServer WebSocket 端点特别简单。你需要编译源文件，并将编译后的类文件包含在 WAR 文件中，最后部署 WAR 文件。Web 容器将检测到 WAR 文件中包含的 WebSocket 端点，并且完成必要的设置和部署工作。一旦完成这些步骤后，就可以首次调用 WebSocket 端点了。我们很快就会看到一些其他的端点，为了部署它们可以有更多的设置工作，但是暂时可以先看看调用刚刚创建的端点的客户端代码。

1.1.3 创建 WebSocket 客户端

在本示例以及本书的众多示例中，我们将用来调用服务器端点的客户端将以利用 JavaScript WebSocket API 的 JavaScript 代码片段的形式存在。这些代码片段将被嵌入 Web 页面，并且被打包在包含 WebSocket 端点的 WAR 文件中。Java WebSocket API 也有客户端 API 的特性，开发人员可以使用 Java WebSocket API 创建应用的客户端，代替使用 Web 页面上的 JavaScript 连接到 WebSocket 服务器端点。第 3 章中将介绍 WebSocket 应用的 Java 客户端。然而，在本示例以及本书中的其他几个示例中，将利用简单的 JavaScript API 进行访问。因为所有支持 WebSocket 的浏览器都支持 JavaScript API，所以它是一个广泛的选择。

在本示例应用中，Web 客户端是一个带有按钮的 Web 页面。当按钮被按下时，将导致 WebSocket 客户端创建一个到 EchoServer 端点的 WebSocket 连接，并发送一条消息。每当 JavaScript WebSocket 接收到一条消息时，它将消息显示在 Web 页面中，这样用户能够看到它，同时也会关闭 WebSocket 连接。在本书后面章节中将能够看到生命周期更长的 WebSocket 连接，目前仅仅简单阐述其第一次交互。

代码清单 1-4 所示是将调用 EchoServer 的 Web 页面的代码，包括最重要的 JavaScript WebSocket 客户端代码。

代码清单 1-4：Echo JavaScript 客户端

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;
      charset=UTF-8">
    <title>Web Socket JavaScript Echo Client</title>
```

```
<script language="javascript" type="text/javascript">
  var echo_websocket;

  function init() {
    output = document.getElementById("output");
  }

  function send_echo() {
    var wsUri = "ws://localhost:8080/echoserver/echo";
    writeToScreen("Connecting to " + wsUri);
    echo_websocket = new WebSocket(wsUri);
    echo_websocket.onopen = function (evt) {
      writeToScreen("Connected !");
      doSend(textID.value);
    };
    echo_websocket.onmessage = function (evt) {
      writeToScreen("Received message: " + evt.data);
      echo_websocket.close();
    };
    echo_websocket.onerror = function (evt) {
      writeToScreen('<span style="color: red;">
        ERROR:</span> ' + evt.data);

      echo_websocket.close();
    };
  }

  function doSend(message) {
    echo_websocket.send(message);
    writeToScreen("Sent message: " + message);
  }

  function writeToScreen(message) {
    var pre = document.createElement("p");
    pre.style.wordWrap = "break-word";
    pre.innerHTML = message;
    output.appendChild(pre);
  }
```

```
    window.addEventListener("load", init, false);

  </script>
 </head>
 <body>
  <h1>Echo Server</h1>

  <div style="text-align: left;">
   <form action="">
    <input onclick="send_echo()" value="Press to send"
                                    type="button">
    <input id="textID" name="message" value=
       "Hello Web Sockets" type="text">
    <br>
   </form>
  </div>
  <div id="output"></div>
 </body>
</html>
```

现在如果运行该应用，将得到如图 1-2 所示的输出。

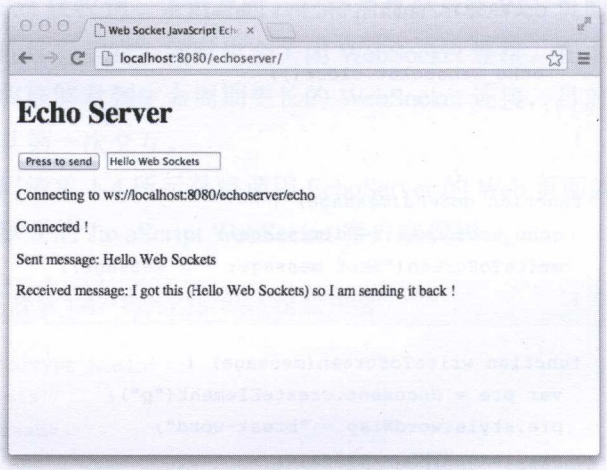


图 1-2　Echo 示例输出

目前为止，你已经创建、部署并且运行了第一个 WebSocket 应用，我们可以不必关注代码，休息一下，转而看看这个非常简单的应用中包含的有关 Java WebSocket API 的基本概念。

1.2　WebSocket 端点

我们使用术语“端点”来表示 WebSocket 对话的一端。当通过@ServerEndpoint 来注解 EchoServer 类时，它将简单传统的 Java 类转换成一个逻辑上的 WebSocket 端点。当部署包含 EchoServer 类的应用时，WebSocket 实现首先扫描包含 EchoServer 类的 WAR 文件，找到其中的 WebSocket 端点。然后，它将此类注册成 WebSocket 端点。当客户端通过 URI /echo 来尝试建立到端点的连接时，因为该 URI 匹配请求中的 URI，WebSocket 实现将创建 EchoServer 类的一个新实例，并使得此 EchoServer 类的实例为后续的 WebSocket 对话服务。当按下示例 Web 页面中的按钮时将调用此实例，其存活时间和 WebSocket 连接的活跃时间保持一致。

注意：

端点是 Java WebSocket API 组件模型的中心：每一个使用 Java WebSocket API 所创建的 WebSocket 应用都有 WebSocket 端点的参与。

目前为止，我们已经了解了使用 Java WebSocket API 创建端点的两种方式中的一种。EchoServer 示例阐述了如何使用 Java WebSocket API 注解将简单 Java 类转换成 WebSocket 端点。第二种方式是继承 Java WebSocket API 中的 Endpoint 抽象类。若如此，此子类将成为 WebSocket 端点，下一节将马上对其进行描述。

注意：

目前存在两种 Java WebSocket 端点：注解式端点和编程式端点。注解式端点是通过 Java WebSocket API 注解将 Java 类转换成 Java WebSocket 端点。编程式端点是通过继承 Java WebSocket API 中的 Endpoint 类来实现。

在讨论什么情况下选择注释式端点和什么情况下选择编程式端点之前，你应该知道，无论采用哪种方法，总的来说都已经可以利用 Java WebSocket API 的大多数特性。

1.3 编程式端点

前面已经介绍过注解式端点，这里主要介绍编程式端点。下一个示例和之前描述过的 EchoServer 示例一样，因为前面已经介绍过，所以你将会很快熟悉它。但它是以编程式端点编写，而不是以注解式端点编写。

为了创建编程式端点类型的 EchoServer 服务器端端点，我们需要做的第一件事情是编写 Java 类，此类继承 Java WebSocket API 中的 Endpoint 类。为了这样做，Endpoint 类要求实现它的 onOpen() 方法。此方法的目的在于，只要客户端连接上所创建的端点实例，就为它注入生命。传入此方法的参数是 Java WebSocket API 的核心类。在详细介绍此 API 之前，先来看看代码。

此处不再列出客户端代码，如果仔细检查编程式 EchoServer 应用的 JavaScript 客户端，你会发现它和注解式 EchoServer 几乎完全一样。唯一不同的地方是它使用了不同的 URI 来连接编程式 EchoServer 端点。

```
ws://localhost:8080/programmaticechoserver/programmaticecho
```

就像我们即将看到的，服务器上的编程式端点映射成/programmaticecho，而不是/echo。

既然明确了这一点，让我们看看代码清单 1-5 所示的编程式端点。

代码清单 1-5：编程式端点示例

```
import java.io.IOException;
import javax.websocket.Endpoint;
import javax.websocket.EndpointConfig;
import javax.websocket.MessageHandler;
import javax.websocket.Session;

public class ProgrammaticEchoServer extends Endpoint {

  @Override
  public void onOpen(Session session, EndpointConfig
     endpointConfig) {
    final Session mySession = session;
    mySession.addMessageHandler
      (new MessageHandler.Whole<String>() {
      @Override
      public void onMessage(String text) {
        try {
          mySession.getBasicRemote().sendText("I got this (
             " +text + ") so I am sending it back !");
        } catch (IOException ioe) {
          System.out.println("oh dear, something went wrong
              trying to send the message back: " +
              ioe.getMessage());
        }
      }
    });
  }
}
```

对于编程式端点首先要注意的是其代码长度比注解式端点要长很多。一些开发人员经常选择创建注解式端点，但是另一些开发人员通常对编程式的处理方式更加熟悉。无论你最终更偏爱哪种处理方式，学习一下此处的示例都是非常有用的，因为即使是如此简单的示例也能够让我们密切接触 API 中的一些关键对象(虽然编程式端点看起来似乎比注解式端点的版本更复杂一些)。两个强制传入 onOpen()方法的对象是 Session 和 EndpointConfig 对象。即使你在创建 WebSocket 应用时经常选择使用注解式方式而不是编程式方式，这些也都是需要理解的非常重要的对象，因为你经常需要它们。

Java WebSocket API 基础对象

Java WebSocket API 中的 Session 对象给了开发人员关于打开的 WebSocket 连接的视图。每一个到 ProgrammaticEchoServer 端点的客户端连接都由 Session 接口的唯一实例表示。它拥有一些方式用于调整连接属性；而且，也许最重要的是，它为端点提供了一个方式来获取对 RemoteEndpoint 对象的访问。RemoteEndpoint 对象表示 WebSocket 对话的另一端，尤其是提供了一种方式使得开发人员可以把消息发送回客户端。

方法 onOpen() 的另一个参数是 EndpointConfig 对象。EndpointConfig接口表示WebSocket实现用来配置端点的一些信息。由于示例中未用到此参数，因此现阶段我们不会讨论此接口。读者如果想了解细节的话，可以参考第 4 章，那里详细介绍了API端点配置。

回到示例程序，onOpen()方法实现做的第一件事情是为会话添加一个MessageHandler对象。MessageHandler接口(及其后代)定义了编程式端点接收入站消息时注册其兴趣的所有方式。例如，

开发人员可以使用MessageHandler接口来选择是接收文本消息还是二进制消息，也可以选择接收整个消息或者是当消息到达时接收较小的片段(这点对交换非常大的消息的应用开发人员特别有用)。示例中实现了最简单的MessageHandler接口：MessageHandler.Whole<String>接口。此接口定义了如何注册一个兴趣来整体接收文本消息。它要求开发人员实现一个单一方法：

```
public void onMessage(String text)
```

此方法在每次客户端的 WebSocket 文本消息抵达时被 WebSocket 实现调用。

示例中，当 WebSocket 实现传递这样的一个文本消息时，我们可以立刻通过调用 Session 对象上的方法：

```
public RemoteEndpoint.Basic getBasicRemote()
```

获得对 RemoteEndpoint 的一个引用。通过使用它可以马上返回一个消息给客户端。

值得注意的是存在两种类型的 RemoteEndpoint：RemoteEndpoint.Basic 和 RemoteEndpoint.Async。RemoteEndpoint 的每个子接口都提供了一种不同的方式来发送消息给它所表示的客户端。RemoteEndpoint.Async接口提供了一系列方法来异步发送消息；也就是说，这些方法初始化消息发送行为，但并不等待消息发送完成就返回。这样，开发人员可以在其应用中忙于做一些其他工作，而不需要阻塞当前工作线程直到消息事实上被发送完毕。简单一些的RemoteEndpont.Basic接口定义了一系列方法用于同步发送消息给客户端；也就是说，每一个send方法调用只有在消息发送完成后才返回。示例中，我们选择使用了此种简单一些的方式。RemoteEndpoint的方法调用发送文本消息给客户端。

```
public void sendText(String text) throws IOException
```

你应该看到它要求开发人员处理受检异常 IOException，此异常在发送消息过程中底层连接出现问题时抛出。

现在我们已经浏览了代码，在编程式端点部署后，当客户端连接到此端点时发生的事情是 WebSocket 实现调用刚刚介绍过的 ProgrammaticEchoServer 类的 onOpen()方法。方法 onOpen()的实现创建 MessageHandler 的实现，此实现在处理客户端消息时，将客户端发送的任何文本消息原样返回。然后，添加 MessageHandler 实例到表示客户端连接的会话上。一旦完成这些工作，此方法就结束了。下次当文本消息从客户端连接到来时，它将被 WebSocket 实现路由到此 MessageHandler 的 onMessage()方法中。

所有的事情似乎都同注解式端点示例完全类似。编程式端点唯一缺少的一部分是没有部署此端点到服务器上的路径。你应该记得注解式端点示例中此路径是注解@ServerEndpoint 的一个属性。在编程式端点的情况中，分派路径稍微复杂一些。为了部署此示例，我们将不得不告诉 WebSocket 实现如何部署此端点。为了做到这一点，我们需要提供 ServerApplicationConfig 接口的实现，它将提供 WebSocket 实现部署端点时需要的这块缺失的信息。

ServerApplicationConfig 接口定义了两个方法来允许开发人员配置应用中的端点。首先来看看第一个方法，因为它相当简单：

```
public Set<Class<?>> getAnnotatedEndpointClasses(
    Set<Class<?>> scanned)
```

此方法在部署应用时被 WebSocket 实现调用。其传入的参数 scanned 是一个集合，它包含所有通过@ServerEndpoint 注解的 Java 类。换句话说，它传入了所有的注解式端点。开发人员实现此方法，通过它返回其实际希望部署的所有的注解式端点的集合。一

般来说，返回的集合将允许 WebSocket 实现部署 WAR 文件中的所有注解式端点。例如，此示例应用中没有注解式端点，所以传入的参数集合 scanned 将为空。

ServerApplicationConfig 接口的第二个方法是我们需要实现用来部署编程式端点的方法：

```
public Set<ServerEndpointConfig> getEndpointConfigs(
    Set<Class<? extends Endpoint>> endpointClasses)
```

同上述注解式端点的方法类似，此方法在应用部署阶段被调用。传入方法的参数是应用中继承了 Endpoint 的所有类的集合。也就是说，它是应用中所有编程式端点的集合。开发人员必须实现此方法，以便其返回 ServerEndpointConfig 对象的集合。此返回的对象集合对应于开发人员希望 WebSocket 实现部署的所有编程式端点。所以在我们的示例中，此方法将被调用并传入一个集合参数，其中的类当然是 ProgrammaticEchoServer 类。这里所需要做的事情是创建 ServerEndpointConfig 对象并将其返回，WebSocket 实现后续将使用此对象部署端点。代码清单 1-6 如下所示。

代码清单 1-6：Echo 示例中的 ServerApplicationConfig

```
import java.util.HashSet;
import java.util.Set;
import javax.websocket.Endpoint;
import javax.websocket.server.ServerApplicationConfiguration;
import javax.websocket.server.ServerEndpointConfiguration;
import javax.websocket.server.
    ServerEndpointConfigurationBuilder;

public class ProgrammaticEchoServerAppConfig
    implements ServerApplicationConfiguration {
  @Override
```

```
    public Set<ServerEndpointConfiguration>
      getEndpointConfigurations(Set<Class<? extends Endpoint>>
        endpointClasses

      ) {
        Set configs = new HashSet<ServerEndpointConfiguration>();
        ServerEndpointConfiguration sec =
          ServerEndpointConfigurationBuilder.create(
            ProgrammaticEchoServer.class, "/programmaticecho")
          .build();
        configs.add(sec);
        return configs;
      }

      @Override
      public Set<Class<?>> getAnnotatedEndpointClasses(
        Set<Class<?>> scanned){
        return scanned;
      }
    }
```

你应该从方法 getAnnotatedEndpointClasses()中可以看到其只是返回了集合 scanned，因为 WAR 文件中不包含注解式端点，所以此示例中该集合是空集。对于方法 getEndpointConfigurations()来说，它首先创建了一个端点配置对象，其拥有希望部署的端点的路径以及端点的实现类。然后，我们将其添加到需要此应用部署的所有的端点配置集合中并返回。它将指示 WebSocket 实现部署此编程式端点到给定的路径/programmaticecho。

然后，我们像部署注解式端点示例一样，将 WAR 文件部署到应用服务器上，而且运行应用时可以得到类似的结果。

也有额外的证据表明，在某种程度上选择创建编程式端点的开发工作量更大。在示例中，为了完成与注解式端点同样的工作，编程式端点本身的代码量更大。当想要部署注解式端点时，可注

意到你无须给出更多的信息或者是编写任何额外的配置代码。然而，当想要部署编程式端点时，则不得不实现一个接口用于提供相关配置信息。

在某种程度上，无论你是否坚持使用注解式端点、编程式端点或者两者混合使用，它仅涉及个人风格。一些开发人员喜欢更传统的显式使用 API 的处理方式。另一方面，一些开发人员更喜欢简洁的注解式处理方式，以及通过改变注解而不是重写配置类而带来的快速改变配置和设置信息的额外灵活性。从功能上来说，虽然有些场景中编程式处理方式比注解式处理方式提供了更多的控制手段和更多的特性，但是两种处理方式基本等价。

本书倾向于使用注解式方式，同时也一定会在示例中混合编程式处理方式。为了能够探索哪种方式将更适合你要编写的应用，以及确定你的个人风格，本书推荐你以开放的心态学习两种方式的基础知识。

1.4　深入 Echo 示例

现在我们已经开发、部署和运行了第一个应用，值得花一些时间通过考察在部署和运行应用时实际发生的事情来理解我们已经做了些什么工作。示例应用中的简单功能暴露了Java WebSocket API中的一些主要元素。因为这些元素几乎在每一个你创建的WebSocket应用中以不同的形式重复出现，因此值得花一些时间来看看这些重要的构建部分。

1.4.1　部署阶段

当我们将包含端点的 WAR 文件部署到应用服务器时，为了

让应用准备好服务其第一个连接，发生了一系列的事情。第一件事情是 WebSocket 实现将检查 WAR 文件，尝试定位可能需要部署的所有端点。首先，此检查将定位被@ServerEndpoint 注解的任何 Java 类和扩展 Java WebSocket API 中 Endpoint 类的任何 Java 类。它也将定位实现了 ServerApplicationConfig 接口的任何类；这些类将告知怎样部署端点。

一旦 WebSocket 实现获取了这些信息，它将使用这些信息来构建一个要部署的端点集合。在 EchoServer 示例中，WAR 文件仅包含一个注解式端点。如果 WAR 文件中不存在 ServerApplicationConfig 接口的实现，WebSocket 实现将自动部署所有的注解式端点。这确实是我们的注解式端点示例的场景，所以那是那个端点如何部署的方式。在编程式 Echo 示例中，WAR 文件包含了一个 ServerApplicationConfig 接口的实现，所以 WebSocket 实现在部署阶段实例化此类一次，并查询其方法来了解哪些端点需要部署。在我们的编程式端点示例中，ServerApplicationConfig 接口要求我们编写的单一的编程式端点被部署。

Java WebSocket API 也有一些开发人员使用的其他部署选项。也可以将多个 ServerApplicationConfig 实现打包在同一个 WAR 文件中。

一旦 WebSocket 实现决定了 WAR 文件中需要部署的 WebSocket 端点集合，多数 WebSocket 实现在进一步处理之前将在端点上运行其他检测，并且仅部署完整的通过检测的应用。不同的 WebSocket 实现在部署阶段可能在不同的地方以不同的方式捕获并且报告可能的配置或者编程错误(例如，两个端点映射到同一个路径或者 WebSocket 注解应用于错误的语义层级)，这些都依赖于其架构和设计。一些 WebSocket 实现可能将故障报告输出到

日志文件。一些实现可能提供图形界面化的工具来输出部署过程中的有帮助的信息。不管怎样，如果一切顺利，应用是有效的，WebSocket 实现将关联这些端点到其声明的 URI 上，然后应用将准备接收入站连接。

1.4.2　接收第一个连接

你应该还记得“前言”中讲到当初始化一个 WebSocket 连接时，不得不发生的第一件事情是最初的 HTTP 请求/响应交互。此交互称为 WebSocket 打开阶段握手(WebSocket opening handshake)。许多 WebSocket 开发人员将永远不需要理解交互工作的细节，就像你不必理解电话交换或者蜂窝网络的机制而能够打电话一样。不管怎样，我们将在第 4 章涉及此主题。在这里，只要说那些希望连接的客户端装备有完整的 URI 地址(包括主机名称以及从主机名到端点发布位置的相对路径)，并且它颁发一个特别构造的 HTTP 请求到那个 URI 就够了。此时此刻，在打开阶段握手的客户端也能够关联其他参数，这是我们第 4 章将讨论的话题。当服务器接收到打开阶段握手请求时，它检查请求，而且可能在客户端执行一系列检查(例如，颁发请求的客户端真的是来自其声称的客户端吗？客户端是否已授权处理？)。当一切顺利时，服务器将返回一个特殊格式化的 HTTP 响应。此响应将告诉客户端服务器是否希望接受客户端的入站连接。在典型场景中，所有这一切发生在“幕后”，远远无需 WebSocket 开发人员的关注。尽管在更高级的场景中，Java WebSocket 开发人员可能拦截此 HTTP 请求和响应交互来进行一些自定义工作。图 1-3 阐述了打开阶段握手的概念。

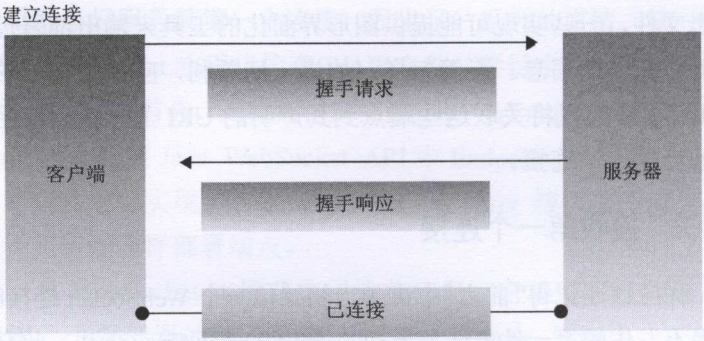


图 1-3　打开阶段握手交互图

如果一切正常，服务器中确实会有一个 WebSocket 端点注册到打开阶段握手提供的地址，建立连接。无论是注解式端点还是编程式端点，WebSocket 实现将创建新的端点实例。此实例将致力于同现在已经连接的单一客户端进行交互。这意味着，如果 WebSocket 端点最终有许多的客户端连接，WebSocket 实现将实例化端点许多次，连接的每个新客户端都会初始化一次。图 1-4 展示了打开阶段握手请求的一个示例。

握手请求

Http 请求

```
GET /mychat HTTP/1.1
Host:server.example.com
Upgrade: websocketConnection:
UpgradeSec-Web
Socket-Key: x3JJHMbDL1EzLkh9GBhXDw==
Sec-WebSocket-Protocol: megachat,chat
Sec-WebSocket-Extensions: compress,mux
Sec-WebSocket-Version: 13
Origin:http://example.com
```

图 1-4　一个打开阶段握手请求

示例中，客户端请求连接托管在http://server.example.com的服

务器上且相对于服务器根的URI路径为/mychat的一个WebSocket。你应该注意到WebSocket协议使用了HTTP升级机制。该机制也被浏览器用于将HTTP连接升级到安全的HTTPS连接，只是示例中客户端请求的协议是WebSocket协议。客户端发送一个唯一标识符，此标识符将在响应中使用。而且你应该注意到图中有一些其他的头定义了客户端希望使用的WebSocket协议的版本，以及希望使用的一些特殊子协议及扩展。我们将在后面章节接触其精确定义和使用WebSocket协议的子协议及扩展。目前，简单地知道它们指示了一种特定应用可以调整协议来更好地适应其需求的方法即可。

最后，打开阶段握手请求可能也声明了 Origin 头。例如，如果请求由浏览器创建，它经常是包含 WebSocket 客户端代码的 Web 页面的网站的因特网地址。

图 1-5 展示了打开阶段握手响应的示例。示例中，服务器同意“升级”连接为 WebSocket 连接。它发送回一个安全标识符，客户端可以使用其来验证它接收的响应是来自于对其发送的同一个请求的回应。示例中的服务器决定使用打开阶段握手请求中列出的一系列协议中的chat子协议。由于它同时支持两种 WebSocket 协议扩展(称为 compress 和 mux——分别表示压缩和多路技术)，因此它也同意在打开阶段握手的客户端和服务器之间已经建立的连接中使用这些扩展。

握手响应

Http 响应

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=
Sec-WebSocket-Protocol: chat
Sec-WebSocket-Extensions: compress, mux
```

图 1-5　打开阶段握手响应

1.4.3 WebSocket 消息通信

现在传输控制协议(TCP)连接已建立。WebSocket 协议在 TCP 之上定义了消息协议的最小帧。这些在 TCP 连接上来回发送的不同的 WebSocket 协议帧定义了 WebSocket 的生命周期事件，例如打开和关闭连接，同时也定义了在连接上应用创建的文本和二进制消息的传输方式。

当 WebSocket 实现接收入站连接时，它保证存在一个可以处理连接的端点的实例，并且将其与发起连接的客户端相关联。在下一章中你将学到更多关于客户端连接与端点的实例相关联的知识。此时，每当客户端发送消息给端点时，都是端点实例的实现在与客户端相关联，接收以负载中消息为参数的方法回调。端点可以从 Session 对象中获取一个到 RemoteEndpoint 对象的引用。此引用唯一表示连接上的客户端，可用于给客户端发送消息。同样的关联也隐式存在于处理注解式端点的消息处理方法返回值的场景中。图 1-6 展示了编程式端点的典型服务器端部署中的对象模型图。

图中，我们可以看到服务器上部署了一个单一的逻辑端点。此端点有两个实例分别由两个 Endpoint 框表示，每个实例处理来自两个独立客户端的消息。每个端点实例已经注册了一系列的 MessageHandler 实例来处理入站消息。你也可以看到每个端点都有一个对表示到每个客户端的连接的 Session 对象的引用。每个会话引用一个 RemoteEndpoint 对象实例，它使得端点具备发送消息给客户端的能力。

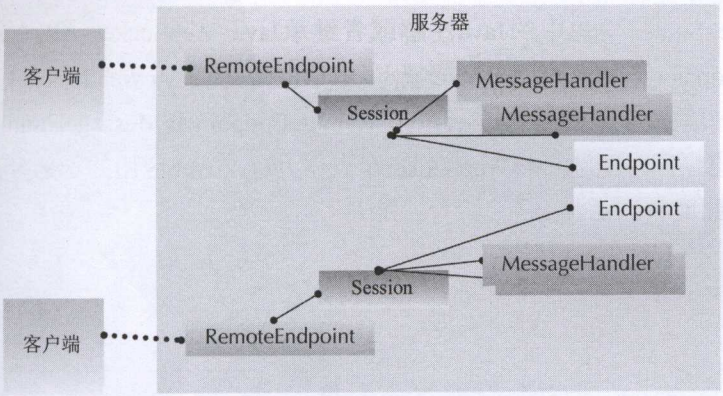


图 1-6　Java WebSocket 服务器应用的对象快照

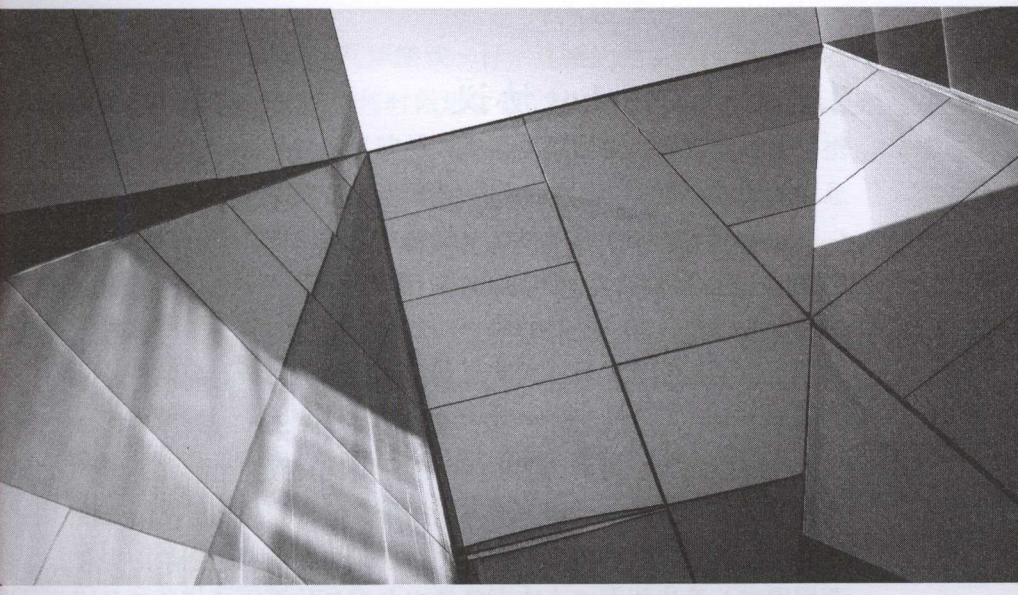
对于注解式端点来说，其运行时建立的对象模型大体相同，端点一直都能够利用 Session 和 RemoteEndpoint 对象。然而，对于注解式端点来说有一个简化，其 MessageHandler 对象由 WebSocket 实现产生；开发人员永远不需要创建、注册或者引用它们。

当然，此图仅仅表示连接存活时的对象的快照。因此 WebSocket 端点的全图就如同两个人在电话中进行谈话的一张照片是一个电话对话的完整描述一样。为了更完整地理解两个端点间完整的 WebSocket 会话，我们需要探究 WebSocket 端点的生命周期。而这将是下一章讨论的主题。

1.5　本章小结

本章中已经看到如何使用Java WebSocket API来创建服务器端的元素并使用JavaScript API来创建客户端元素，从而创建简单的客户端/服务器WebSocket应用。同样也看到如何使用Java

WebSocket API中的Java注解或者继承Java WebSocket API中的Endpoint类来创建WebSocket端点。本章也提及了Java WebSocket API中一些最重要的对象：Session、RemoteEndpoint和MessageHandler对象。同时阐述了将WebSocket端点作为标准Web应用的一部分部署的两种方法。



第2章

Java WebSocket 生命周期

本章将阐述 WebSocket 端点的生命周期。WebSocket 端点的生命周期为开发人员提供了一个框架来管理端点所需的资源，也提供了一个框架来拦截消息。我们将仔细查看其生命周期事件的顺序和语义，以及 Java WebSocket API 如何提供 API 和注解来支持处理这些事件。我们将看到如何在示例应用中以注解式和编程式两种方式来使用它们。

2.1 WebSocket 协议

我们首先介绍 WebSocket 协议本身的背景。为了能够使用 Java WebSocket API，读者无须知晓本节的每一个细节，但是它能作为有益的背景资料帮助理解 WebSocket 技术以及 Java WebSocket API 形成的原因。

与基于 HTTP 的技术不同，WebSocket 具有生命周期。此生命周期由 WebSocket 协议进行支撑。例如，在 Servlet 技术中，底层协议仅仅定义了简单的请求/响应交互，此交互完全独立于下一次交互。事实上，在大部分情况下携带交互数据的底层网络连接将被完全弱化。一些技术(例如 Java Servlet API)必须在请求/响应交互模型之上构建会话，这有助于开发人员创建比单一的隔离的交互生存时间更长的应用：HttpSession 和 Java Servlet 组件生命周期就是其最佳示例。

相反，WebSocket 协议定义了客户端和服务器间长时间存活的专用的 TCP 连接的正因为如此，在定义更长时间的生命周期方面，它比传统的 Web 请求/响应模型更进一步。此外，WebSocket 协议定义了 WebSocket 连接上往返传输的数据的各个块的格式。一旦连接已经建立，这些传输的元数据帧描述了其用途。WebSocket 协议中包含两种主要类型的帧：控制帧和数据帧。控制帧是用于执行协议的一些内部功能逻辑的数据传输。例如，协议定义了关闭帧(close frame)。关闭帧是一种特殊的传输，它意味着发送者准备关闭连接。其他控制帧是 Ping 帧和 Pong 帧。Ping 帧和 Pong 帧是用来服务于连接健康性检测的数据传输。如果 WebSocket 希望检查其到 WebSocket 对等节点的连接的健康性，它可以发送 Ping 帧。当 WebSocket 对等节点收到 Ping 帧后，必

须响应 Pong 帧。同乒乓球游戏一样，连接的健康性(多快或者其功能是否正常)可以在任何时候进行检测。

WebSocket 协议定义的另一类帧是数据帧。数据帧定义了携带应用数据的 WebSocket 传输的种类。第 1 章中发送的回显消息就是文本数据帧的示例。数据帧分为两种基本类型：文本型和二进制型，一种用来携带文本消息，另一种用来携带二进制数据(例如图像数据)。WebSocket 协议的一个特性是它能够通过多个独立的传输来发送消息，不管消息是文本类型还是二进制类型。这样的单独传输携带了完整消息的一部分，同时也作为此传输序列的一部分，其被称为部分帧(partial frame)。此种将消息作为部分帧序列的技术在 WebSocket 实现传递特别大的消息时或者是发送正在制定中的消息时特别有用。虽然此协议经常仅用来发送短消息，但是它并没有限制 WebSocket 消息的大小，所以也有一些应用选择使用 WebSocket 消息来传输大量的数据，例如视频或者金融数据的大型存档文件。

你将看到，低层级的 WebSocket 协议帧的提示在一些 Java WebSocket API 调用中是很明显的，然而开发人员实际上不需要理解其低层级的细节。

2.2　Java WebSocket 生命周期

既然你已经对 WebSocket 会话期间连接上往返交互的过程有了一定的了解，下面就来看看 Java WebSocket API 如何对端点生命周期进行建模。

所有 Java WebSocket 端点生命周期的第一个事件是打开通知(open notification)，它用来指示到 WebSocket 会话另一端的连接已

经建立。使用电话呼叫来作一个类比，在第 1 章中已经谈到此握手交互与因特网电话呼叫建立过程中发生的交互过程类似：电话号码路由、选择使用的连接速度，以及是否仅使用语音还是同时使用视频连接。打开通知类似于电话铃声响，你接听电话并单击建立打开的连接。

一旦打开通知被 WebSocket 对话的两端都接收到，参与的任意 WebSocket 后续就可以发送消息了。发送多少消息、何时发送、发送顺序、发送内容当然都高度依赖于应用。与打电话类似，打电话期间，任何一方都可以有机会在电话中说其想说的话。

在WebSocket对话期间的任何时候，都可能出现一些消息传递的错误。接收消息的WebSocket端点本身就可能会产生错误(例如，它接收消息但不知如何合理地处理消息)，或者WebSocket实现本身在某些情况下也会产生错误(例如，接收的消息比它能够处理的消息还要大)。WebSocket端点生命周期的这一阶段可能会产生两种结果：错误是致命错误，这将导致连接被关闭，无法再传递消息；错误是非致命错误，此时端点将有能力自己决定是否继续发送和接收消息。和打电话类似，消息可能来自对话的任意一端，它可能暂时被干扰，但随后又变得连续，或者一些更严重的错误可能事实上导致对话被中断。

不管 WebSocket 对话的哪端准备结束对话，它都可以初始化关闭事件。在这种情况下，WebSocket 实现与对话的另一端进行通信，通知其连接将被关闭。和打电话类似，这个过程类似于电话中的一方准备说“再见”来结束对话。事实上，WebSocket 无须等待另一方确认此通知事件，它类似于对话伙伴已经说了再见但没来得及以同样的方式响应时立刻听到了电话滴答声。或许对打电话的人来说这种行为有些无礼，但是对于 WebSocket 端点来说却完全合适。

借助于图 2-1，我们回顾一下 WebSocket 对话的生命周期。在这里我们可以自顶向下看看这个图，一旦客户端和服务端的 WebSocket 连接被建立，WebSocket 对话的两端都被通知会话打开了。一旦会话被打开，WebSocket 对话的两端会接收到其对方发来的消息。

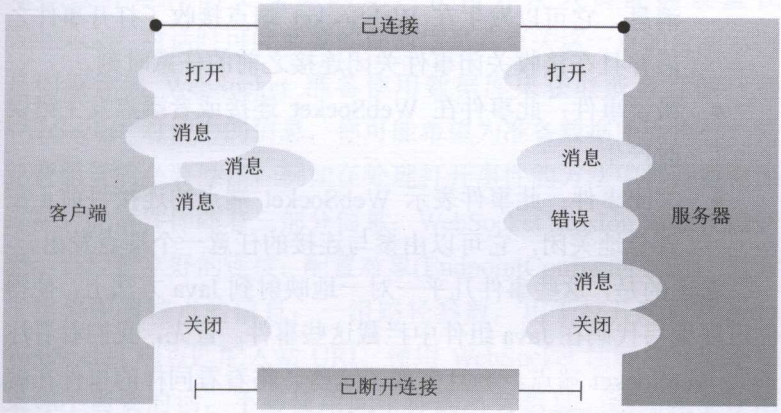


图 2-1　WebSocket 对话生命周期

在此阶段中，如图中的情况下，可能出现的一些错误状态是可恢复的。因为服务端端点在接收错误并且在处理完错误之后发送了另一条消息。图中，WebSocket 对话的一端已经决定结束对话，将导致两端接收到关闭通知(close notification)。此后除非建立新的连接，否则客户端和服务端间不会再发送消息。

2.3　Java WebSocket API 中的 WebSocket 生命周期

既然已经基本了解了 WebSocket 端点生命周期，下面就从

Java 组件的视角来看看其生命周期是如何呈现的。在上一节中，我们已经识别了 WebSocket 端点的如下 4 个生命周期事件。

- 打开事件：此事件发生在端点上建立新连接时并且在任何其他事件发生之前。
- 消息事件：此事件接收 WebSocket 对话中另一端发送的消息。它可以发生在 WebSocket 端点接收了打开事件之后并且在接收关闭事件关闭连接之前的任意时刻。
- 错误事件：此事件在 WebSocket 连接或者端点发生错误时产生。
- 关闭事件：此事件表示 WebSocket 端点的连接目前正在部分地关闭，它可以由参与连接的任意一个端点发出。

幸运的是，这些事件几乎一对一地映射到 Java 方法上，使得你可以编写代码在 Java 组件中拦截这些事件。首先，我们看看注解式 WebSocket 端点。一旦完成了这些，将看看同样的事件在编程式端点上的展现方式。

2.3.1 注解式端点事件处理

为了将Java类声明成WebSocket端点，对于服务器端端点来说需要使用一个类级别注解@ServerEndpoint(第1章中已经描述过)，对于客户端端点来说需要使用类似的@ClientEndpoint注解(将在第3章中描述)。现在假设我们正在开发一个端点，此端点将部署在服务器上并且将等待来自一个或者更多客户端的入站连接。

对于注解式端点来说，为了拦截不同的生命周期事件，我们需要利用方法级注解：@OnOpen、@OnMessage、@OnError和@OnClose。我们将看到每个WebSocket生命周期事件都伴随着不同的限定信息。因此，对于被生命周期注解注解的Java方法可能的方法签名，有许多选项。

1. @OnOpen

首先介绍第一个注解@OnOpen。此注解用于注解式端点的方法，指示当此端点建立新的连接时调用此方法。需要一个方法来处理打开事件的主要原因是，使得开发人员能够设置在 WebSocket 对话时可能需要的任何信息。有关这一点的合适示例是如果你的 WebSocket 准备使用数据库来获取或者存储一些 WebSocket 对话中的信息，你可能希望为准备数据库而执行一些花费昂贵的必要操作，例如在处理打开事件的方法中打开数据库连接。此事件伴随着三部分信息：WebSocket Session 对象，用于表示已经建立好的连接；配置对象(EndpointConfig 的实例)，包含了用来配置端点的信息；一组路径参数，用于打开阶段握手时 WebSocket 端点匹配入站 URI。使用 WebSocket 注解的好处是如果你不需要的话，不必使用此事件的所有信息。因此此类使用@OnOpen 注解的方法是任何没有返回值的公有方法，这些方法有一个可选的 Session 参数、一个可选的 EndpointConfig 参数，以及任意数量的被@PathParam 注解的 String 参数。另外，这些参数的顺序可以任意排列。我们将在第 6 章回到路径参数主题，因此此处不在其上花费过多时间。这意味着如果仅仅希望能够引用 Session 对象，可以编写如下的代码清单 2-1。

代码清单 2-1：打开事件处理方法示例

```
@OnOpen
public void init(Session session) {
    // initialization code
}
```

此外，如果你希望查询配置对象(由 EndpointConfig 类的实例表示)，同样可以进行如代码清单 2-2 所示的声明：

代码清单 2-2：打开事件处理方法示例

```
@OnOpen
public void init(Session session, EndpointConfig config) {
    // initialization code
}
```

你已经看到，不仅能够灵活地命名方法来处理打开事件，而且能够灵活地决定事件关联的数据中有多少可用。

2. @OnMessage

通常，你应该希望 WebSocket 端点能够在连接建立后处理一些或者所有的入站消息。为此，我们使用@OnMessage 注解。此注解允许你装饰你希望处理入站消息的方法。Java WebSocket API 中的消息事件伴随的信息是 Session 对象(它表示消息抵达时的连接)、EndpointConfig 对象、打开阶段握手中从匹配入站 URI 过程中获取的路径参数以及最重要的消息本身。与@OnOpen 注解一样，方法参数可以以任意顺序排列，并且你仅需要包含你想知道的消息事件的那些部分的方法参数。

连接上的消息将以 3 种基本形式抵达：文本消息、二进制消息或 Pong 消息。Java WebSocket API 提供了一系列的选项使得能够以这些形式接收消息。最基本的形式是选择使用带 String 参数的方法来处理文本消息；使用带 ByteBuffer 或者是 byte[]参数的方法来处理二进制消息；若你的消息将仅仅处理 Pong 消息，则可使用 Java WebSocket API 中的 PongMessage 接口的一个实例。

例如，如果你希望以最简单的形式来处理文本消息，可以使用如下所示的代码清单 2-3：

代码清单 2-3：文本消息处理方法示例

```
@OnMessage
public void handleTextMessages(String textMessage) {
    // process the textMessage here
}
```

如果你希望处理二进制消息，并且希望获取表示消息到来时连接的会话的引用，可使用如下所示的代码清单 2-4：

代码清单 2-4：二进制消息处理方法示例

```
@OnMessage
public void processBinary(byte[] messageData, Session session) {
    // process binary data here
}
```

尽管如此，对于文本和二进制消息来说有一些更加高级的选项。例如，你可以选择当消息到来时分批地接收文本或者二进制消息。若如此，可以使用一对参数来表示到来的消息分片(partial message)，如代码清单 2-5 所示。对于文本消息分片来说是 String 和 boolean，如代码清单 2-6 所示。String 表示文本消息分片，boolean 是一个标志位，当其被设置为 false 时表示后续还有整个文本消息序列中的更多的消息分片到来，当设置为 true 时表示当前的消息分片是序列中的最后一个消息分片。对于二进制消息分片来说，可以选择一对参数 ByteBuffer 和 boolean，其中 ByteBuffer 表示二进制消息分片，boolean 表示到来的此二进制消息分片是否为构成完整消息的序列中的最后一个消息分片。对于二进制消息来说，也可以使用一对参数 byte[]和 boolean，它与参数对 ByteBuffer 和 boolean 一样，但其使用字节数组而不是 ByteBuffer 来存储二进制消息分片。

代码清单 2-5：二进制消息分片处理方法示例

```
@OnMessage
public void processVideoFragment(byte[] partialData,
   boolean isLast) {
 if (!isLast) {
    // there is more to come !
 } else {
    // now we have the whole message !
 }
}
```

代码清单 2-6：文本消息分片处理方法示例

```
@OnMessage
public void catchDocumentPart(String text, boolean isLast) {
   // pass on to feed elsewhere
}
```

即使使用此消息分片处理选项，无论使用哪种处理方式，短消息仍然可能以一个片段到达。长消息可能以任意数量的片段到达。一般说来，它取决于 WebSocket 实现如何将消息发送到 API：不同的实现可能会将消息拆分成更小一些的片段。尽管如此，一般说来如果你希望能够在消息一到来时就开始处理，则它是处理大消息的一个有用选项。

使用@OnMessage 注解来处理输入消息的另一个选项是选择使用 Java I/O 流：使用 java.io.Reader 来处理文本消息，使用 java.io.InputStream 来处理二进制消息。如代码清单 2-7 所示。试图使用一些使用了 I/O 的 Java 类库来处理消息时，它们是有用的。

代码清单 2-7：二进制 I/O 消息处理方法示例

```
@OnMessage
public void handleBinary(InputStream is) {
```

```
    // read
}
```

事实上，处理消息还有更多的选项：你甚至可以让 WebSocket 实现把入站消息转换成自己选择的对象。我们将在第 3 章回到此话题，描述更多的细节。

WebSocket 应用一般是异步的双向消息。换言之，典型应用并不总是立即响应入站消息。尽管如此，在一些场景下你希望立刻响应入站消息：第 1 章中的 Echo 示例应用就是关于这一点的极好示例。因此，通过@OnMessage 注解的此类方法上有一个额外选项：方法可以有返回类型或者返回为空。当使用@OnMessage 注解的方法有返回类型时，WebSocket 实现立即将返回值作为消息返回给刚刚在方法中处理的消息的发送者。它在你需要设计一个应用显式确认收到 WebSocket 消息的特殊情况下是有用的。如代码清单 2-8 所示。

代码清单 2-8：包含返回值的消息处理方法示例

```
@OnMessage
public byte[] dealWithRequest(String requestMessage) {
  byte[] ack = {0};
  return ack;
}
```

在示例中，当拥有此方法的端点接收任意类型的文本消息时，端点立刻返回包含 0 个字节负载的二进制消息。通过@OnMessage 注解的方法的返回类型中最基本的类型是 String、byte[]或者 ByteBuffer。还有更多的选项，具体可参见下一章。

3. @OnError

错误事件比消息事件稍微简单一些。@OnError 可以用来注解

WebSocket 端点的方法，使其可以处理 WebSocket 实现处理入站消息时发生的任何错误。如代码清单 2-9 所示。如果你不想让 WebSocket 端点处理这些类型的错误的话，就可以不处理。尽管如此，还是建议在 WebSocket 端点中包括处理错误的方法，否则你在试图追踪客户端发送了但没有传达的消息时可能以浪费时间而结束。伴随错误事件的信息是错误信息、发生错误的会话以及与建立连接的打开阶段握手相关联的任何一个路径参数。再次重申，我们将在下一章回到路径参数的主题。错误信息由 java.lang.Throwable 类表示，会话由 Java WebSocket API 中的 Session 接口表示。和其他 Java WebSocket API 中的方法级注解一样，错误处理方法中包含哪些参数依赖于错误发生时你希望方法所接收的信息。与其他方法级注解一样，方法中的参数顺序可以任意排列。

代码清单 2-9：错误处理方法示例

```
@OnError
public void errorHandler(Throwable t) {
  // log error here
}
```

处理入站消息时，可能会发生 3 种基本的错误类型。首先，WebSocket 实现产生的错误可能会发生，例如如果该 WebSocket 端点的入站消息是非正常的。这些错误都属于 SessionException 类型。其次，错误可能发生在当 WebSocket 实现试图将入站消息解码成开发人员所要求的某个对象时。我们将在第 3 章回到此话题，此处只需要知道此类型错误都是 DecodeException 就足够了。最后是由 WebSocket 端点的其他方法产生的运行时错误。

如果你选择不在应用中包含错误处理方法，那么 WebSocket

实现将记录 WebSocket 端点操作过程中产生的任何错误，以便后续查看。当然，从哪里能够找到这些信息以及信息是否足够详细都依赖于所使用的 WebSocket 实现。

4. @OnClose

现在我们来看看 WebSocket 生命周期中的最后一个事件：关闭事件。如果你在 WebSocket 对话中使用了一些昂贵的资源并且希望释放和恰当地关闭它们，拦截关闭事件是一个很好的主意。它对于在 WebSocket 连接关闭时做其他的通用清理工作也是有用的。@OnClose 可以用来注解多种不同类型的方法来处理关闭事件。如代码清单 2-10 所示。伴随关闭事件的信息是关闭信息、与建立连接的打开阶段握手相关联的任意一个路径参数，以及一些描述连接关闭原因的信息。最后的关闭原因信息以 Java WebSocket API 中 CloseReason 类的形式存在。连接关闭存在多种原因，但是最典型的原因是 WebSocket 对话的任意一方完成了所需要的所有工作或者是由于不活跃导致连接超时。按惯例，方法参数都是可选的，而且可以以任意顺序出现。

代码清单 2-10：关闭事件处理方法示例

```
@OnClose
public void goodbye(CloseReason cr) {
   // log the reason for posterity
   // close database connection
}
```

现在，我们将所有这些概念放在一起，以示例的形式解释这些主要概念。

2.3.2 Lifecycle 示例

Lifecycle 示例是一个 JavaScript WebSocket 客户端。它与 Java WebSocket 端点进行交互，并且使用所有的生命周期注解。当 LifecycleEndpoint 注解式端点运行于服务器上处理 WebSocket 生命周期事件时，它发送消息给客户端使得客户端能够显示一系列交通信号灯。当应用启动时，连接是关闭的。点击 Web 页面上的按钮将一步步地带着用户通过 WebSocket 生命周期中的关键阶段，如图 2-2 所示。你将注意到有两个按钮能够关闭连接：一个按钮导致客户端初始化关闭事件；另一个按钮导致服务器初始化关闭事件。

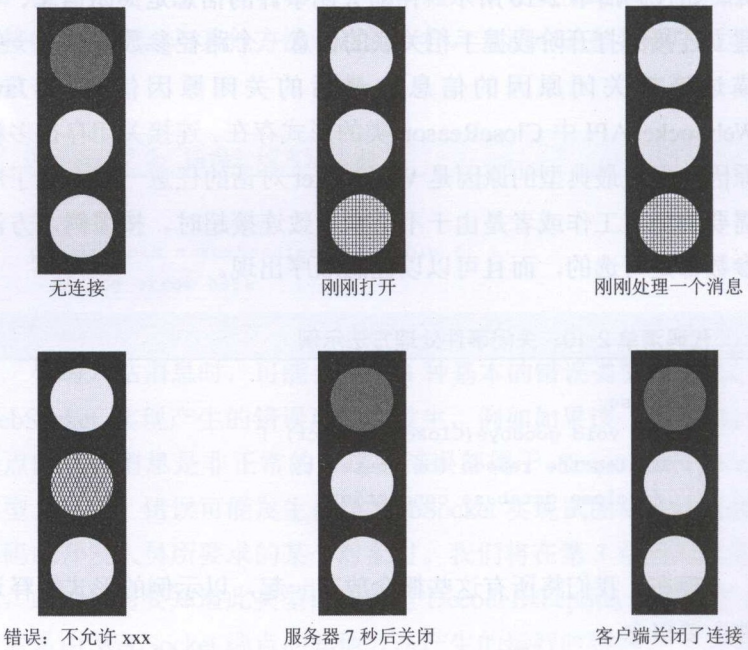


图 2-2 交通信号灯生命周期

1. Lifecycle JavaScript 客户端

JavaScript 代码使用 JavaScript WebSocket API 拦截来自 Java 类 LifecycleEndpoint 发送的消息，并解析消息来告知哪个信号灯亮着以及需要给用户显示什么消息。

首先来看看 JavaScript 代码，仅仅关注主要的客户端方法调用即可。按惯例，如代码清单 2-11 所示，我们创建了 JavaScript WebSocket 实例，为其添加事件处理代码使得每当服务器组件发送回消息时都将被调用，并将指示客户端如何显示交通信号灯。

代码清单 2-11：创建一个 JavaScript WebSocket

```
function open_connection() {
 lifecycle_websocket = new
    WebSocket("ws://localhost:8080/lifecycle/lights");
 lifecycle_websocket.onmessage = function (evt) {
      update_for_message(evt.data);
      update_buttons();
 };
 lifecycle_websocket.onclose = function (evt) {
      update_buttons();
 };
}
```

如你所见，当 JavaScript WebSocket 接收到消息时，它将调用方法 update_for_message()并且更新按钮的启用/禁用状态。另外，当 JavaScript WebSocket 关闭时，你将看到也会调用同样的方法更新按钮状态。如代码清单 2-12 所示。

其次，我们有两个方法来解析来自服务器的消息。这些消息的一个示例是“3：Just opened”。第一个方法去除消息部分，仅返回数字；第二个方法去除开始的数字，仅返回消息部分。

代码清单 2-12：解析生命周期消息的 JavaScript

```
function get_light_index(message) {
  return message.substring(0, 1)
}

function get_display_message(message) {
  return message.substring(2, message.length)
}
```

下一个方法基于其在交通信号灯中的位置(light_index)和应该打开还是关闭(light_on_index)来计算交通信号灯中每个灯的颜色。如代码清单 2-13 所示。

代码清单 2-13：计算交通信号灯颜色的 JavaScript

```
function get_color(light_index, light_on_index) {
  if (light_index == 1 && light_on_index == 1) {
    return "red"
  } else if (light_index == 2 && light_on_index == 2) {
    return "yellow"
  } else if (light_index == 3 && light_on_index == 3) {
    return "green"
  } else {
    return "grey"
  }
}
```

最后，这里有一个方法将所有的一切放在一起：它被调用时的参数是服务器发送的消息中解析出来的部分，并且其目的是更新交通信号灯以及信号灯下的消息。如代码清单 2-14 所示。

代码清单 2-14：响应生命周期消息更新页面的 JavaScript

```
function update_display(light_index, display_message) {
  var old = traffic_light_display.firstChild;
```

```
    var pre = document.createElement("pre");
    pre.style.wordWrap = "break-word";
    pre.innerHTML = "<b><font
        face='Arial'>"+display_message+"</font></b>";
    if (traffic_light_display.firstChild != null) {
        traffic_light_display.replaceChild(pre,
            traffic_light_display.firstChild);
    } else {
        traffic_light_display.appendChild(pre)
    }
    var context = document.getElementById
        ('myDrawing').getContext('2d');
    context.beginPath();
    context.fillStyle = "black"
    context.fillRect(65,0,70,210);
    context.fill();

    context.beginPath();
    context.fillStyle = get_color(1, light_index);
    context.arc(100,35,25,0,(2*Math.PI), false)
    context.fill();

    context.beginPath();
    context.fillStyle = get_color(2, light_index);
    context.arc(100,105,25,0,(2*Math.PI), false)
    context.fill();

    context.beginPath();
    context.fillStyle = get_color(3, light_index);
    context.arc(100,175,25,0,(2*Math.PI), false)
    context.fill();
}
```

我们注意到此示例中使用了新的绘画画布。它和 WebSocket 一样，也是属于 HTML5 规范的一部分。

2. Lifecycle 注解式端点

目前我们已经了解了应用客户端所做的工作，下面回到 WebSocket 端点生命周期的主题。让我们来看看我们创建的服务器组件 Lifecycle 注解式端点。代码清单 2-15 如下所示。

代码清单 2-15：Lifecycle Java WebSocket 端点

```
import java.io.*;
import java.io.IOException;
import javax.websocket.OnClose;
import javax.websocket.OnError;
import javax.websocket.OnMessage;
import javax.websocket.OnOpen;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;

    @ServerEndpoint("/lights")
public class LifecycleEndpoint {
     private static String START_TIME = "Start Time";
     private Session session;

     @OnOpen
     public void whenOpening(Session session) {
       this.session = session;
        session.getUserProperties().put(START_TIME,
            System.currentTimeMillis());
       this.sendMessage("3:Just opened");
     }

     @OnMessage
     public void whenGettingAMessage(String message) {
       if (message.indexOf("xxx") != -1) {
         throw new IllegalArgumentException("xxx not
             allowed !");
       } else if (message.indexOf("close") != -1) {
         try {
```

```
            this.sendMessage("1:Server closing after "
                + this.getConnectionSeconds() + " s");
            session.close();
        } catch (IOException ioe) {
            System.out.println("Error closing session "
                + ioe.getMessage());
        }
        return;
    }
    this.sendMessage("3:Just processed a message");
}

@OnError
public void whenSomethingGoesWrong(Throwable t) {
    this.sendMessage("2:Error: " + t.getMessage());
}

@OnClose
public void whenClosing() {
    System.out.println("Goodbye !");
}

void sendMessage(String message) {
    try {
        session.getBasicRemote().sendText(message);
    } catch (Throwable ioe) {
        System.out.println("Error sending message "
            + ioe.getMessage());
    }
}

int getConnectionSeconds() {
    long millis = System.currentTimeMillis()
        - ((Long) this.session.getUserProperties().
            get(START_TIME));
    return (int) millis / 1000;
}
}
```

首先，我们注意到此服务器端点使用了@ServerEndpoint注解，它被映射到路径/lights上。你可能想知道在不看需要它来初始化连接的客户端代码的情况下如何计算此端点的完整URI。此完整URI是ws:// URI，其中URI由主机名称和端口号，加上包含此WebSocket端点的Web应用的上下文路径，再加上此端点的相对URI构成。我们将在第6章回到此话题，然而由于此处包含Lifecycle端点的Web应用的上下文路径是/lifecycle，因此意味着此WebSocket端点的完整URL是：

```
ws://localhost:8080/lifecycle/lights
```

你将看到示例中用@OnOpen注解指定的方法用来处理WebSocket端点的打开事件，代码清单 2-16 如下所示。

代码清单 2-16：Lifecycle 示例中打开事件处理方法签名

```
@OnOpen
public void whenOpening(Session session)
```

我们已经选择此方法接收一个 Session 对象的引用，此对象将在后面使用。通过查看此方法的实现，我们将看到端点中的此方法在接收到打开事件之后所做的事情是记录当前的时间并将其添加到 Session 对象的用户属性字典中，然后给客户端发送消息指示将第三个灯(绿灯)打开来表明消息通信现在可以流动了。

你在示例中也会看到每当接收到消息时，都将调用如代码清单 2-17 所示的方法。

代码清单 2-17：Lifecycle 示例中消息事件处理方法签名

```
@OnMessage
public void whenGettingAMessage(String message)
```

我们已经选择仅接收文本消息(任何抵达此端点的二进制消息将被忽略)并且以 String 对象的形式接收。与前面谈到的选择以消息分片或者是以 java.io.Reader 形式接收文本消息相比，由于此处的消息都是短消息，此形式是完全合适的。事实上，在很多场景中，很可能你会选择以这种最简单的形式来接收文本消息。此消息事件处理方法的实现中首先检查消息是否有不良字符的特殊序列(毕竟这些交通信号灯是在一个家族中)。如果发现不正确，将产生一个包含解释的运行时异常。如果发现正确，方法将检查消息查看客户端是否请求服务器端端点关闭连接，此种场景下方法将在给客户端发送一条指示将灯变红的消息后关闭连接。最后，如果消息是其他种类的消息，此方法的实现将发送一个消息给客户端，指示信号灯应该继续保持绿色。

如果消息中有不良字符，它将创建一个需要被处理的错误事件。因此 Lifecycle 端点声明如代码清单 2-18 所示的方法来处理任何错误事件。

代码清单 2-18：Lifecycle 示例中错误事件处理方法签名

```
@OnError
public void whenSomethingGoesWrong(Throwable t)
```

此方法简单地给客户端发送一条消息，此消息包含错误描述并且请求客户端将交通信号灯变黄。当运行此应用时，如果你发送一条坏消息，Lifecycle 示例中的消息处理方法将被调用。这将产生一个异常，它接着导致错误处理方法被调用，并接着将交通信号灯中的黄灯点亮。因为错误事件并没有导致会话被关闭，黄灯意味着消息通信仍将流动，在这种场景下是合适的。当然，在错误处理方法中也可以做一些其他的事情，例如调用 session.close()方法来关闭连接。

最后，每当客户端连接关闭时，都将调用 Lifecycle 端点声明的如代码清单 2-19 所示的方法。

代码清单 2-19：Lifecycle 示例中关闭事件处理方法签名

```
@OnClose
public void whenClosing()
```

你将注意到无论是 Lifecycle 客户端还是服务器端点终止连接，都将调用 LifecycleEndpoint 类的 whenClosing()方法。

2.3.3 编程式端点生命周期

在继续讨论此示例的其他方面之前，简单地看一下编程式 WebSocket 端点生命周期。

1. 生命周期事件

你应该记得第 1 章中提到过编程式端点都必须继承 javax.websocket.Endpoint 类。可以通过提供 Endpoint 方法的自定义实现来拦截编程式端点的打开、关闭和错误事件。既然你已经学习了 WebSocket 生命周期注解，那么应该相当熟悉 Endpoint 类的如表2-1所示的方法。

表 2-1 端点方法

事 件	端 点 方 法
打开	public abstract void onOpen(Session ses, EndpointConfig config)
错误	public void onError(Session ses, Throwable thr)
关闭	public void onClose(Session ses, CloseReason cr)

事实上，编程式端点经常需要处理的唯一事件是打开事件，

它提供了对会话和配置信息的访问。编程式端点不需要拦截其他任何生命周期事件。

2. 处理消息

到目前为止你可能想知道如何拦截消息事件，当然你已经看到这一点。因为在 Java WebSocket API 中开发人员有广泛的选项可以用来处理消息，因此此工作有一些轻微的差别。

如第1章所见，为了处理入站消息，你需要提供MessageHandler实现。如同注解式端点，这里存在广泛的选项用于创建不同的MessageHandler实现来以不同的方式处理不同类型的WebSocket消息。我们将在第3章中查看这些参数的全集，现在仅关注处理基本文本和二进制消息。对此仅需要实现下列这些：对于文本消息，使用MessageHandler.Whole<String>；对于二进制消息，使用MessageHandler.Whole<ByteBuffer>。

一旦你实现了上述一个或者两个接口来定义希望消费消息的方式，你需要做的所有事情是通过调用

```
session.addMessageHandler(myMessageHandler)
```

在第一个消息到达之前的某一时刻注册你的消息处理程序到代表你有兴趣侦听的连接的 Session 对象上。通常，端点将在 onOpen() 方法中添加其消息处理程序，因此可以确保不遗漏任何消息。

3. 编程式 Lifecycle

代码清单 2-20 所示是 Lifecycle 端点的编程式版本：

代码清单 2-20：编程式 Lifecycle WebSocket 端点

```
import java.io.IOException;
```

```
import javax.websocket.CloseReason;
import javax.websocket.Endpoint;
import javax.websocket.EndpointConfig;
import javax.websocket.MessageHandler;
import javax.websocket.Session;

public class ProgrammaticLifecycleEndpoint extends Endpoint {
  private static String START_TIME = "Start Time";
  private Session session;

  @Override
  public void onOpen(Session session, EndpointConfig config) {
    this.session = session;
    final Session mySession = session;
    this.session.addMessageHandler(
        new MessageHandler.Whole<String>() {
      @Override
      public void onMessage(String message) {
        if (message.indexOf("xxx") != -1) {
          throw new IllegalArgumentException(
          "xxx not allowed !");
        } else if (message.indexOf("close") != -1) {
          try {
            sendMessage("1:Server closing after " +
               getConnectionSeconds() + " s");
            mySession.close();
          } catch (IOException ioe) {
            System.out.println("Error closing session "
            + ioe.getMessage());
          }
          return;
        }
        sendMessage("3:Just processed a message");
      }
    });
    session.getUserProperties().put(START_TIME,
    System.currentTimeMillis());
    this.sendMessage("3:Just opened");
```

```
    }

    @Override
    public void onClose(Session session, CloseReason closeReason) {
      System.out.println("Goodbye !");
    }

    @Override
    public void onError(Session session, Throwable thr) {
      this.sendMessage("2:Error: " + thr.getMessage());
    }

    void sendMessage(String message) {
      try {
        session.getBasicRemote().sendText(message);
      } catch (IOException ioe) {
        System.out.println("Error sending message " + message);
      }
    }

    int getConnectionSeconds() {
      long millis = System.currentTimeMillis() -
        ((Long) this.session.getUserProperties().
        get(START_TIME));
      return (int) millis / 1000;
    }
}
```

2.3.4　实例数目及线程机制

此示例中，你可能想知道的一件事情是为什么端点使用实例变量来保存对Session对象的引用。你应该记得Session对象表示到单一客户端的连接。你应该也注意到在Lifecycle示例中，我们在端点的生命周期中的两个地方使用了会话对象：一个地方是在客户端建立新连接时用于记录事件发生的时间，另一个地方是在方

法whenGettingAMessage()中当服务器端点准备关闭到客户端的连接时计算连接打开的时间。

因为可以很容易地如代码清单 2-21 所示编写关闭事件处理方法，所以我们无须在打开事件处理方法中将会话存储为实例变量以便后续访问。

代码清单 2-21：另一种 Lifecycle 关闭事件处理方法签名

```
@OnClose
public void whenGettingAMessage(String message,
Session session)
```

我们可以简单地要求 WebSocket 实现传入这些参数。

Lifecycle 示例中将会话存储为实例变量的原因是，我们可以使用它来阐述 WebSocket 端点生命周期中的一个更重要的问题。如果你重新启动 Lifecycle 应用，但是这一次打开第二个浏览器窗口到同样的首页，你将看到两组交通信号灯。假如你开始按下任意一个浏览器窗口的生命周期按钮，那么将看到每组信号灯都可以是不同的状态。这是因为每个浏览器窗口对 Lifecycle WebSocket 端点来说都充当一个独立的客户端，并且 WebSocket 实现为每个连接的客户端使用不同的 LifecycleEndpoint 实例。

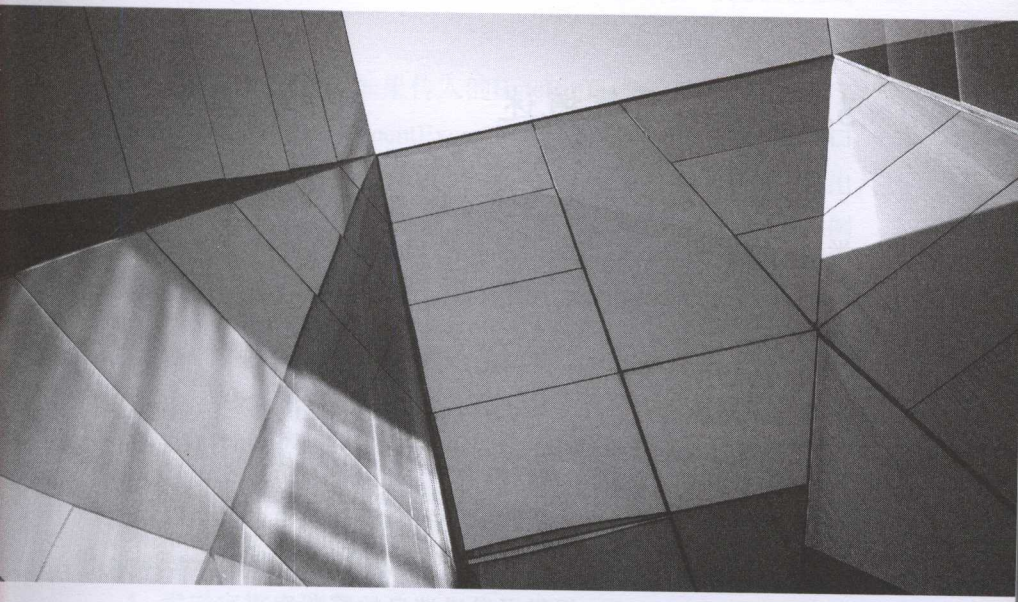
这意味着对于每一个 WebSocket 端点(不管是注解式 Java 类还是编程式端点)定义来说，WebSocket 容器在每次有新的客户端连接时会实例化端点的一个新的实例。这样做的结果是每个 WebSocket 端点实例仅能够永远“看到”同样的会话实例：此实例表示从唯一的客户端连接到那个端点实例的唯一连接。

同样，可以编写 Lifecycle 示例将会话对象作为参数传递给 whenGettingAMessage()方法。若如此，你将发现传递的实例与传递给 whenOpening(Session session)方法的 Session 对象是一样的。

WebSocket 实现也为你提供了另外一个重要的保证：同一个会话(或者是连接)中不允许两个事件线程同时调用一个端点实例。这可能听起来很抽象，但是这意味着端点实例永远不会在某时被 WebSocket 实现的一个以上的线程调用。它意味着如果客户端发送多条消息，WebSocket 实现必须调用端点每次处理一条消息。知道这一点特别重要，因为这意味着你永远不需要担心为端点实例的并发访问进行编程。这也是 Java WebSocket 编程模型与 Java Servlet 编程模型的关键差异，Java Servlet 实例可能被多个线程同时调用，每个线程用于处理不同客户端的请求/响应交互。这意味着 WebSocket 编程明显更加容易。

2.4　本章小结

本章中学习了管理 WebSocket 端点生命周期的事件。不仅学习打开、消息、错误和关闭事件的顺序和语义，而且也学习了那些允许开发人员处理这些事件的方法和注解。最后，还学习了 WebSocket 实现调用开发人员创建的 WebSocket 端点时所使用的线程策略，它意味着 WebSocket 开发人员通常不需要处理多个线程同时对端点的调用。



第3章

消息通信基础

本章主要介绍 Java WebSocket API 在消息通信方面的基本概念。在 Java WebSocket API 中包含了很多发送和使用消息的方式，本章主要关注消息发送中最直接也是最普遍的方式：消息的同步发送和接收。为了能阐明覆盖到的这些概念和功能，后面会深入介绍一个 DrawingBoard 应用。该应用使用 Java 客户端前端，因此与此同时也会介绍 Java WebSocket API 的客户端 API。

3.1 消息通信概述

在第 2 章中提到过在 WebSocket 协议中定义了 3 种本地消息类型：文本消息、二进制消息以及 Ping 和 Pong 消息。让我们回顾一下 Java WebSocket API 中用于发送和接收消息的所有必要部分。

3.1.1 发送消息

为了发送 WebSocket 消息，无论创建的是一个注解式 WebSocket 端点还是一个编程式 WebSocket 端点，其 API 都是相同的：RemoteEndpoint 接口和它的子类(RemoteEndpoint.Basic 和 RemoteEndpoint.Async)提供了发送消息的所有方法。

Ping 和 Pong 消息通常被开发人员用来检查 WebSocket 底层连接的健康性，这样做有很多理由：可以检查 WebSocket 连接是否还有效，或者可以通过测量 Ping 和 Pong 消息所花费的时间来测算 WebSocket 连接的效率。现在，可以说很多应用并不依赖这类消息。然而，如果需要发送这类消息，从 Session 对象获取到的 RemoteEndpoint 实例提供了代码清单 3-1 所示的两个方法。

代码清单 3-1：RemoteEndpoint 发送 Ping 和 Pong 消息

```
public void sendPing(ByteBuffer applicationData)
  throws IOException, IllegalArgumentException

public void sendPong(ByteBuffer applicationData)
  throws IOException, IllegalArgumentException
```

WebSocket规范定义了Ping和Pong消息可以传输125个字节大小的二进制数据：这就是上述每个方法都用了ByteBuffer参数的原

因。调用这些方法时，如果传入的ByteBuffer参数超过了125个字节，则会抛出IllegalArgumentException异常。RemoteEndpoint接口和它的派生类中的所有发送消息的方法都会通过抛出IOException异常用来表明在消息的传输过程中出现了错误。例如，如果在调用发送消息之前底层连接已断开，那么就会发生此类异常。

现在转向更常见的消息类型：文本消息和二进制消息。在第 5 章中，我们将介绍 Java WebSocket API 中关于异步发送消息的 API 和语法。现在，我们还是限于介绍消息的同步发送，也就是说当前我们需要重点学习 RemoteEndpoint.Basic 接口，到第 5 章时再介绍 RemoteEndpoint.Async 接口。

1. 发送字符串消息

RemoteEndpoint.Basic API 提供了 3 种发送字符串的方法。第一个同时也是最简单的方法如代码清单 3-2 所示。

代码清单 3-2：RemoteEndpoint.Basic 发送文本消息

```
public void sendText(String text) throws IOException
```

该方法把传入的文本参数作为 WebSocket 文本消息发送，这意味着 WebSocket 连接的另一端将收到一个文本格式的消息。只有在消息被成功发送或者在发送过程中抛出了错误时，该方法才返回，如在发送过程中连接被断开了。在第 1 章和第 2 章中已经见过该方法的使用，它是发送 WebSocket 消息的最基本也是最常用的方式。

由于 WebSocket 消息通常表现为一些高层级的对象形式(序列化成 String 以便发送)，因此 Java WebSocket API 也提供了一种

使用 Writer API 发送 String 消息的方式，代码清单 3-3 如下所示。

代码清单 3-3：RemoteEndpoint.Basic 发送文本消息到流

```
public Writer getSendStream() throws IOException
```

当使用其他 API(提供了把字符数据写入 Writer 的方法)时，这个方法特别有用，例如 javax.swing.text.html.HTMLEditorKit API 就允许把一段 HTML 文档写入 Writer 中。

现在，WebSocket 协议允许把大的 WebSocket 消息分解成多个小片段。这使得 WebSocket 实现可以在整个消息被完全传送之前开始发送消息，并获得性能优化，这对于那些发送非常大的消息的应用来说特别明显。为了让开发人员能够利用消息分片传递的优势，Java WebSocket API 也提供了如代码清单 3-4 所示的方法。

代码清单 3-4：RemoteEndpoint 以片段形式发送文本消息

```
public void sendText(String partialMessage, boolean isLast)
throws IOException
```

该方法允许开发人员以小片段序列的形式发送大的字符串消息。调用该方法时，isLast 参数一般设置为 false，直到发送最后一个片段时才将其设置为 true，以表明整个消息发送完毕。

2. 发送二进制消息

对于大多数应用来说，发送文本形式的消息已经足够。然而，对于那些有特殊数据格式(如小图像文件)的应用或者要求绝对最小消息长度的应用来说，以二进制形式发送消息是个非常好的选择。幸运的是，Java WebSocket API提供了多种方式用于发送二进制格式的消息。正如在介绍文本消息时所做的一样，在这里我们

将只关注使用RemoteEndpoint.Basic接口同步发送消息，而把使用RemoteEndpoint.Async接口异步发送消息的介绍放在第 5 章。

有 3 种发送二进制消息的方法，先看其中的两种，如代码清单 3-5 所示。

代码清单 3-5：RemoteEndpoint 发送二进制消息

```
public void sendBinary(ByteBuffer data) throws IOException

public void sendBinary(ByteBuffer partialByte,
boolean isLast) throws IOException
```

ByteBuffer 参数用于传输消息数据，如果遇到了阻止消息数据发送的通信错误，则会抛出 IOException 异常。跟文本消息一样，第一个方法是一次性发送完整的消息。第二个方法则是通过发送一个或多个二进制消息片段来完成消息发送。该方法中的 isLast 参数用于指明要发送的消息片段是否是整个消息的最后一个，或者指明是否还有消息片段会到达。当消息非常大或者消息产生是一个逐步的过程时，用第二个方法是个很好的选择。

注意：

对于二进制和文本消息，在某些应用中，以消息片段序列的形式发送消息能够获得非常大的性能优化。然而，消息最终以什么形式进行传输取决于 WebSocket 实现。对于一个大消息，有些实现可能用大量的小片段来传输，而有些实现可能用少量的大片段来传输。有些实现可能需要你来配置，还有些实现甚至需要调整本地网络条件的参数。因此，在选择 WebSocket 实现时，以及在你的程序中该方法达到最高效率之前，做一些小的实验和测试是非常值得的。

最后，可以得到一个用来写入二进制消息数据的java.io.OutputStream的引用。这非常有用，特别是当使用直接将数据对象写入Java I/O的API时。在完成消息写入后需要关闭输出流。一旦关闭输出流，消息就会被发送。为了能用这种方式发送二进制消息，需要使用如代码清单 3-6 所示的方法。

代码清单 3-6：RemoteEndpoint.Basic 使用流发送二进制消息

```
public OutputStream getSendStream() throws IOException
```

3. 发送 Java 对象消息

现在，你已经领会到有很多方法可以用来发送文本或者二进制消息，这对很大一类应用来说已足够。这些应用倾向于发送相对简单的消息，如简单的股票更新、温度测量或者新闻摘要等。但是，也有很多 WebSocket 应用需要发送更加复杂的消息。这些消息可能包含数据、指令或者授权，也可能包含结构化程度很高的数据。对于这类应用，能把这些消息模型化会变得很有益。这样的消息不仅仅是文本或者是二进制数据，还可以是高层次的 Java 对象。

在 Java WebSocket API 中，可以使用 RemoteEndpoint.Basic 发送任意 Java 对象消息，方法如代码清单 3-7 所示。

代码清单 3-7：RemoteEndpoint.Basic 发送 Java 对象

```
public void sendObject(Object data) throws IOException,
EncodeException
```

你可能很想知道WebSocket实现是如何把对象变成WebSocket消息的(正如在第 2 章中读到的，WebSocket只能发送 3 种基本消息：文本消息、二进制消息以及Ping和Pong消息)。答案取决于你

传什么类型的对象。

如果你传一个 Java 基本类型(或者其等值装箱类)，则 WebSocket 实现会把数据转换成一个标准的 Java 字符串(就是使用 toString()方法)。

如果传入的是其他对象，那么需要为 WebSocket 实现提供一个 javax.websocket.Encoder 接口的实现。在本章后面会看到一个示例，在这里先快速了解下。在 Encoder 家族中，最通用的接口是 javax.websocket.Encoder.Text<T>，T 就是你想发送的对象的类型。接口的主要方法如代码清单 3-8 所示。

代码清单 3-8：Encoder.Text<T>接口的编码方法

```
public String encode(T object) throws EncodeException
```

每次使用 RemoteEndpoint.Basic 的 sendObject 方法发送 T 类型的对象时，WebSocket 实现都会调用相应的编码器。发送给远程端点的实际上是 encode()方法返回的字符串。如果你的编码器无法把指定对象转换成字符串，很可能会抛出 EncodeException 异常。在这种情形下，EncodeException 将会被传播给 RemoteEndpoint.Basic 的 sendObject()方法。

还可以灵活选择其他的 Encoder 接口。例如，如果想把对象编码成 WebSocket 二进制消息，可以实现 Encoder.Binary<T>接口。如果想把对象编码成 Java I/O 流，可以实现 Encoder.CharacterStream<T> 或者 Encoder.BinaryStream<T>。

一旦创建好了要使用的编码器，就需要选择如何在端点上配置它。

对于注解式端点，所有需要做的就是在类级别的 WebSocket 注解上声明 Encoder 类。假设有一个 Orchard 应用，需要发送一条

类型是"Apple"的对象消息，那么就需要在服务器端点上配置一个能对Apple对象进行编码的编码器，很可能需要用到如代码清单3-9所示的注解。

代码清单 3-9：在@ServerEndpoint 注解中指定编码器

```
@ServerEndpoint(
  value = "/fruit_trees",
  encoders = { MyAppleEncoder.class }
)
```

一旦这样做，每当你从这个注解式端点上获取一个RemoteEndpoint 引用时，都可以直接传一个 Apple 对象给sendObject()方法，WebSocket 实现会使用 MyAppleEncoder 把Apple 对象编码成 WebSocket 消息。

如果使用编程式端点，为了部署端点，你需要创建EndpointConfig 对象，在创建该对象时可以配置编码器类。例如，如果要创建一个需要使用 MyAppleEncoder 编码器的编程式客户端端点，会需要代码清单 3-10 所示的代码。

代码清单 3-10：为编程式客户端端点配置编码器

```
List<Class<? extends Encoder>> encoders = new ArrayList<>();
encoders.add(MyAppleEncoder.class);
ClientEndpointConfig config =
  ClientEndpointConfig.Builder.create()
                              .encoders(encoders)
                              .build();
```

正如你将在本章后面的 DrawingBoard 应用中看到的一样，为了部署客户端端点，需要创建一个 ClientEndpointConfig 实例。

现在，每当你从这个编程式端点获取一个 RemoteEndpoint 引

用时，都可以调用它的 sendObject()方法，传入 Apple 实例。WebSocket 实现会使用你提供的 MyAppleEncoder 实现把 Apple 实例编码成 WebSocket 消息。

我们以表格形式来结束这段关于如何通过 RemoteEndpoint 接口发送自定义 Java 对象消息的介绍。表 3-1 总结了所有你能选择用来实现和配置端点的编码器接口类型。

表 3-1　编码器接口类型

编码器接口	转　换	主 要 方 法
Encoder.Text<T>	T 转换成 String	String encode(T Object) throws EncodeException
Encoder.TextStream<T>	T 转换成 Writer	void encode(T object, Writer writer) throws EncodeException, IOException
Encoder.Binary<T>	T 转换成 ByteBuffer	ByteBuffer encode(T object) throws EncodeException
Encoder.BinaryStream<T>	T 转换成 OutputStream	void encode(T object, OutputStream os) throws EncodeException, IOException

3.1.2　接收 WebSocket 消息

我们已经了解了在 WebSocket 应用中发送消息的一些方法，接下来看看 WebSocket 端点可以选择的接收消息的方法。

1. 在注解式端点中接收 WebSocket 消息

在第 1 章中我们看到了注解式端点接收文本消息的一种最简

单的方法，就是写带 String 参数的方法，并且在该方法上标记@OnMessage 注解，如代码清单 3-11 所示。

代码清单 3-11：用@OnMessage 处理文本消息

```
@OnMessage
public void handleTextMessages(String textMessage) {
  return "I got this " + textMessage + " !");
}
```

这是在端点中可以选择用于接收文本消息的最简单但不是唯一的方式。通过使用合适的方法参数，还能选择接收二进制 WebSocket 消息和 Pong 消息。如代码清单 3-12 所示，为了接收二进制消息，最容易的方式就是使用 byte[]数组参数。

代码清单 3-12：用@OnMessage 处理二进制消息

```
@OnMessage
public String handleBinaryMessages(byte[] messageData) {
  return "I got " + messageData.length + " bytes of data !";
}
```

为了接收 Pong 消息，只需要声明一个类型为 javax.websocket.PongMessage 的方法参数。如代码清单 3-13 所示。

代码清单 3-13：用@OnMessage 处理 Pong 消息

```
@OnMessage
public String handlePongMessages(PongMessage pongMessage) {
  return "I got a pong message carrying " +
    pongMessage.getApplicationData().length +
            " bytes of data !");
}
```

这 3 个方法变体覆盖了 WebSocket 的两个主要消息类型(文本

消息和二进制消息)，以及如何捕获入站的 Pong 消息。

注意：

侦听到 Ping 消息时会发生什么？此刻你可能想知道，为什么我们不介绍侦听入站 Ping 消息的方法？答案是 Java WebSocket API 没有提供这样的方法。WebSocket 实现被要求以最快的速度回复连接中入站的任何 Ping 消息(例如你在 RemodeEndpoint 上使用 sendPing()方法手动发送了一个 Ping 消息)，Pong 消息包含的数据与 Ping 消息相同。因此，不需要显式地写代码来侦听 Ping 消息。

与发送和处理文本消息以及二进制消息的方法变体很类似，有很多可以选择用来接收它们的方法变体。就像发送消息一样，可以以消息片段序列的形式来接收消息。例如，如果你知道 WebSocket 会话的另一端要发送一个巨大的消息，WebSocket 实现很可能会把大消息分解成一系列的小消息。在这种情况下，你可以选择在消息片段到达时就逐个接收，而不是等到整个消息传完后再接收。在这种情形下，需要声明一个类似代码清单 3-14 所示的方法。

代码清单 3-14：用@OnMessage 处理到达的文本消息片段

```
@OnMessage
public void handlePartial(String textMessagePart,
boolean isLast)
```

每当一个新消息片段到达时，WebSocket 实现会调用这个方法。textMessagePart 参数是消息片段的内容，boolean 类型的 isLast 标志位在消息未发送完全时会被置为 false，当最后的消息片段到达时置为 true。正如即将看到的，二进制消息也可以使用类似的方式。

表 3-2 给出了一个清单，为 Java 开发人员列出了注解式端点方法如何以最熟悉的形式接收入站的文本消息、二进制消息以及 Pong 消息。

表 3-2 接收方法参数类型

参 数 类 型	处理的消息类型	示 例
String	文本消息	public void handle(String message)
String、boolean	文本消息片段	public void handle(String partialMessage, boolean isLast)
Reader	文本消息流	public void handle(Reader message)
byte[]	二进制消息	public void handle(byte[] data)
ByteBuffer	二进制消息	public void handle(ByteBuffer data)
byte[]、boolean	二进制消息片段	public void handle(byte[] partialData, boolean isLast)
ByteBuffer、boolean	二进制消息片段	public void handle(ByteBuffer partialData, boolean isLast)
PongMessage	Pong 消息	public void handle(PongMessage message)

之前有提到过，很多 WebSocket 应用需要处理比字符串或者非结构化数据要更复杂的数据对象。值得庆幸的是，Java WebSocket API 也为这类应用提供了支持，允许开发人员接收任意形式的 Java 对象消息。如果开发人员提供了一个解码器类，那么 WebSocket 实现就知道如何把收到的 WebSocket 消息转换成开发人员要求的 Java 对象。例如，如果你想接收一个 Orange 对象形式的消息，就需要在注解式端点上写下类似代码清单 3-15 所示的方法。

代码清单 3-15：用@OnMessage 处理 Java 对象消息

```
@OnMessage
public void addToBasket(Orange orange) {
  this.bag.addShoppingItem(orange);
  this.cost = this.cost + orange.getPrice();
}
```

为了使这段代码工作起来，WebSocket 实现必须知道如何将入站消息转换成一个 Orange 对象，为此你必须提供一个 Decoder 接口的实现。与 Encoder 接口类似，有大量可用的子类型供你实现。现在，我们将通过查看 Decoder.Text<T>接口来说明其基本概念，Decoder.Text<T>控制 WebSocket 实现如何把入站的文本消息转换成 T 类型的 Java 对象。在我们的示例中，主要需要实现如代码清单 3-16 所示的 Decoder.Text<Orange>接口的 decode()方法。

代码清单 3-16：Decoder.Text<Orange>接口的 decode()方法签名

```
public Orange decode(String rawMessage) throws DecodeException
```

该接口还需要实现代码清单 3-17 所示的 willDecode()方法。

代码清单 3-17：Decoder.Text<Orange>接口的 willDecode()方法签名

```
public boolean willDecode(String s)
```

在 WebSocket 实现中，该方法先于 decode()方法被调用，这是为了使你有一个跳过解码消息的机会，例如当消息格式明显不正确时。

解码器接口有许多的子类型，用于区分需要转换成 Java 对象的 WebSocket 消息类别以及转换方式。接下来的表 3-3 汇总了所有的解码器接口，当你想以 T 类型对象的形式接收消息时，就会用到它们。

表 3-3 解码器接口

解码器接口	转　换	主要解码方法
Decoder.Text<T>	String 转换成 T	T decode(String raw) throws DecodeException
Decoder.TextStream<T>	Reader 转换成 T	T decode (Reader raw) throws DecodeException
Decoder.Binar	ByteBuffer 转换成 T	T decode (ByteBuffer raw) throws DecodeException
Decoder.BinaryStream	InputStream 转换成 T	T decode (InputStream raw) throws DecodeException

你会发现，如果想解码的原始数据无法转换成期望的 Java 对象，将会抛出 DecodeException 异常。在所有这些解码失败中，引起失败的入站消息不会被传递，但是解码器中产生的 DecodeException 异常会被传递给端点的错误处理方法。

提示：

在你的 WebSocket 端点中，应始终包含错误处理方法。除了处理入站消息的错误之外，端点上的其他 WebSocket 方法(如打开事件处理方法)产生的运行时异常也会被传递到这里。如果没有错误处理方法，你可能不知道消息是否已经到达过。

Java WebSocket API 提供了一个方便，为 Java 基本类型和它的等价类提供了内置文本解码器。WebSocket 实现采取的途径是使用基本类型或者等价类的标准 Java 字符串表示。这就意味着解码操作等同于 Java 基本类型转换成其等价类，使用单个字符串参数的构造函数生成等价类。例如，如果开发人员写了一个方法希

望以 Java 整型形式处理入站的文本消息，如代码清单 3-18 所示。

代码清单 3-18：用@OnMessage 处理值为整型的 WebSocket 消息

```
@OnMessage
public void doCount(Integer message) {
   // process Integer
}
```

当值为 42 的文本消息到达时，会调用上述的 doCount()方法处理，Integer 对象的值等于 new Integer(42)。

下面的表 3-4 总结了 Java 对象消息的传递选项。这些消息不是之前介绍的文本消息、二进制消息以及 Ping 和 Pong 消息，而是Java基本类型及其等价类，或者是开发人员自定义的Java对象。

表 3-4　Java 对象消息的传递选项

传递选项	解码器	示　例
Java 基本类型及其等价类的文本消息	自动	@OnMessage public void handleTransferCode(Double d)
自定义 Java 对象的文本或者二进制消息	开发人员提供	@OnMessage public void handleObject(CustomObject o)

在离开注解式端点接收消息的主题之前，我们需要覆盖最后一个主题，该主题与那些以@OnMessage 注解的方法密切相关。

到目前为止的所有示例中，@OnMessage方法的返回类型都是void。然而，你会记得在第 1 章的Echo示例中，有一个非常方便的方法用来立即响应入站的WebSocket消息。那个注解了@OnMessage的方法提供了返回值，返回值的类型决定了WebSocket实现要寄回给消息发送者的WebSocket消息的类型。

为了能回应一个文本消息，返回类型应为 String；为了回应

一个二进制消息，返回类型应为 byte[]或者 ByteBuffer。最后，通过使用在本章中已讨论过的转换规则，还可以响应标准 Java 基本类型及其等价类的文本消息。这意味着在注解式端点中，代码清单 3-19 所示的所有方法都是有效的消息处理方法。

代码清单 3-19：消息处理方法示例

```
@OnMessage
public String echo(String message) { ... }
@OnMessage
public Integer processAndConfirm(byte[] upload) { ... }
@OnMessagepublic boolean purchase(String item) { ... }
```

多个消息处理方法

我们已经看到了在注解式 WebSocket 端点中有很多种方式来消费入站的消息，你可能很想知道，如果端点中有多个处理方法时会发生什么。

在注解式端点上，Java WebSocket 实现为了能将入站消息分配到正确的消息处理方法上，它设置了一个非常严格的限制：每个注解式端点最多只有一个消息处理方法处理每种本地 WebSocket 消息类型(即文本消息、二进制消息和 Pong 消息)。

这样引起的结果是，比如不能有两个注解了@OnMessage 的方法都希望处理文本消息，再比如不能有类似代码清单 3-20 所示声明的注解式端点。

代码清单 3-20：多消息处理方法的非法使用

```
@ServerEndpoint(
  value="/orchard",
  decoders= {OrangeDecoder.class}
)
public class FruitTree {
```

```
    @OnMessage
    public void handleString(String message) {
        ...
    }
    @OnMessage
    public void handleOrange(Orange orange) {
        ...
    }
}
```

OrangeDecoder 实现了 Decoder.Text<Orange> 或者 Decoder.TextStream<Orange>的任意一个。这个示例违反了最多只能有一个方法用来处理每类本地WebSocket消息的规则，因为有两个方法都能处理入站的文本消息。如果尝试部署这样的端点，你会发现这样做会导致部署错误，端点不会被成功部署。为了让上述示例变成一个有效的注解式端点，需要把这两个消息处理方法拆分到两个不同的注解式端点中去。

2. 在编程式端点中接收 WebSocket 消息

现在，我们对在注解式端点上如何以多种形式处理入站的 WebSocket 消息有了一定的理解，接下来看看如何在编程式端点中处理入站消息。

编程式端点使用什么方式接收消息取决于 MessageHandler 接口和它的子类型。例如，为了以 Java String 对象的形式接收文本消息，回顾第 1 章的编程式 Echo 示例可知我们需要实现 MessageHandler. Whole<String>接口，并在编程式端点的 onOpen 方法中把消息处理程序添加给当前 Session 对象。如代码清单 3-21 所示。

代码清单 3-21：在编程式端点中处理文本消息

```
public void onOpen(Session session, EndpointConfig
    endpointConfig) {
  final Session mySession = session;
  mySession.addMessageHandler(new
    MessageHandler.Whole<String>() {
    @Override
    public void onMessage(String text) {
      handleTextMessage(text);
    }
    });
  }

private void handleTextMessage(String text) {
    // handle incoming messages here
}
```

正如你所期望的一样， Java WebSocket API 提供了大量方法用于在编程式 WebSocket 端点中处理入站消息。无论选择哪一种方法，都需要遵循以下两个步骤：

(1) 实现 MessageHandler 的某个子类，这取决于你要处理的消息类型以及你想如何处理它。可以选择接收整个 WebSocket 消息或者是把 WebSocket 消息变成消息片段的序列来接收。

(2) 在编程式端点的当前 Session 对象上配置 MessageHandler 实现。通常，当使用 onOpen()方法创建新会话时就需要这样做，尽管在端点生命周期的其他点上也可以交换 MessageHandler 实现。

下面的表 3-5 列出了 MessageHandler 的所有变体，描述了它们能消费的入站消息类型，以及它们会把消息如何呈现给编程式端点。

表 3-5　MessageHandler 的变体

MessageHandler	消 息 类 型	传 递 形 式
MessageHandler.Whole<String>	文本	Java String 对象
MessageHandler.Whole<Reader>	文本	Java I/O 流
MessageHandler.Whole<ByteBuffer>	二进制	Java NIO ByteBuffer
MessageHandler.Whole<byte[]>	二进制	Byte 数组
MessageHandler.Whole<InputStream>	二进制	Java I/O 流
MessageHandler.Partial<String>	文本片段	Java 字符串序列
MessageHandler.Partial<ByteBuffer>	二进制片段	ByteBuffer 序列
MessageHandler.Partial<byte[]>	二进制片段	字节数组序列

这里看上去很熟悉，是因为在编程式端点中消费传入 WebSocket 消息的选项和在注解式端点中是一样的。

与注解式端点一样，可以选择以 Java 对象的形式接收入站的 WebSocket 消息。如果要这样做，则需要提供一个能将本地 WebSocket 消息转换成 Java 对象的 Decoder 接口实现。因此，在上表中我们追加如下的选项，见表 3-6。

表 3-6　MessageHandler 的其他变体

MessageHandler	消 息 类 型	传 递 形 式
MessageHandler.Whole<T>和 Decoder.Text<T>或者 Decoder.TextStream<T>	文本	对象类型 T
MessageHandler.Whole<T>和 Decoder.Binary<T>或者 Decoder.BinaryStream<T>	二进制	对象类型 T

与在编程式端点上配置Encoder实现相似，为了支持MessageHander接收自定义对象，必须在部署的时候把你提供的所有Decoder实现配置好。例如，为了把OrangeDecoder(之前在注解式端点示例中用到过)配置到以WAR文件形式部署的编程式端点上，我们可以使用如代码清单3-22所示的ServerApplicationConfig实现的示例：

代码清单 3-22：在编程式服务器端点中配置 Decoder

```
public class MyServerApplicationConfig implements
ServerApplicationConfig {

  public Set<ServerEndpointConfig> getEndpointConfigs(
    Set<Class<? extends Endpoint>> endpointClasses) {
    Set<ServerEndpointConfig> configs = new HashSet<>();
    List<Class<? extends Decoder>> decoders = new ArrayList<>();
    decoders.add(MyOrangeDecoder.class);
    ServerEndpointConfig config =
      ServerEndpointConfig.Builder.create(MyEndpoint
        .class, "/fruit")
        .decoders(decoders)
        .build();
    configs.add(config);
    return configs;
  }

  public Set<Class<?>> getAnnotatedEndpointClasses(
    Set<Class<?>> sc) {
    return sc;
  }
}
```

我们最终以一段警告来结束在编程式端点中如何选择接收WebSocket消息的学习。对于编程式端点的每个Session的每个本地WebSocket消息类型，Java WebSocket API只允许一个

MessageHandler，以便WebSocket实现在WebSocket消息到达时能够轻松地决定调用哪个MessageHandler。对开发人员来说，这样的后果就是调用下面的方法时：

```
session.addMessageHandler(myMessageHandler)
```

如果已经为 myMessageHander 消费的 WebSocket 消息类型(文本消息、二进制消息或者 Pong 消息)注册了其他的 MessageHandler，那么该方法会抛出 java.lang.IllegalStateException 异常。

你可能已经注意到这个限制与之前在注解式端点中提到过的情形是对应的，即每一个注解式端点只允许最多一个方法来处理每个基本 WebSocket 消息类型。

注意：

为了能确定某个特定 MessageHandler 所消费的基本 WebSocket 消息类型，你需要查看配套的 Decoder，看它是解码文本消息还是二进制消息，或者两个都解码。

3.2 DrawingBoard 应用

在本节中，我们以示例应用的形式将已经介绍过的概念和细节串联起来。

DrawingBoard 示例是一个多客户端/单服务器的 Java WebSocket 应用。该示例的客户端是一个 Java Swing 应用(包含了一个客户端 Java WebSocket 端点)，它呈现了一个简单的画布，用户可以在画布上画各种各样不同颜色或大小的图形。DrawingBoard 应用的服务器端是一个 Web 应用，包含了单个服务器端 Java WebSocket 端点。每当用户在画布上画图形时，客户端

向服务器发送更新。当服务器接收到更新时，它会把更新的特征通知给所有已连接的客户端。客户端从服务器获取更新内容，然后更新画布，反映在原绘画客户端中作的更改。在这种方式下，所有已连接的客户端实际上是在团体合作绘画。图 3-1 显示了两个客户端窗口正在合作画一个田园风光图。

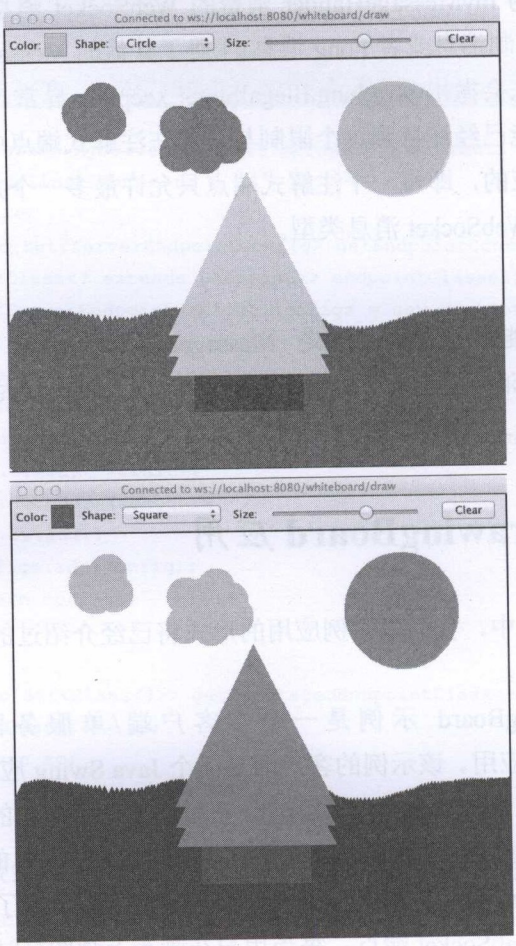


图 3-1　两个 DrawingBoard 用户的协作

DrawingBoard 客户端

我们首先看一下客户端 WebSocket 端点。到目前为止，我们所有的注解式端点都是服务器端的端点，因此该 DrawingBoard 客户端端点是我们第一次学习客户端 WebSocket 端点。客户端端点使用@ClientEndpoint 代替@ServerEndpoint。这个类级别的注解是客户端的，与@ServerEndpoint 注解对等。一旦使用它注解了一个 Java 类，正如服务器端注解式端点一样，便能使用任意或者全部方法级别的注解去拦截生命周期事件和 WebSocket 消息。

在学习客户端端点之前，我们先看一下应用中用于描述绘画对象的代码，它们被描述成 DrawingObject 和 Shape 这两个类。代码清单 3-23 所示是 DrawingObject 类。

代码清单 3-23：DrawingObject 类

```
import java.awt.*;

public class DrawingObject {
  public static String MESSAGE_NAME = "DrawingObject";
  private Shape shape;
  private Point center;
  private int radius = 0;
  private Color color;

  public DrawingObject(Shape shape, Point center, int radius,
    Color color) {
    this.shape = shape;
    this.center = center;
    this.radius = radius;
    this.color = color;
  }

  public Shape getShape() {
```

```
    return this.shape;
  }

  public Point getCenter() {
    return this.center;
  }

  public int getRadius() {
    return this.radius;
  }

  public Color getColor() {
    return this.color;
  }

  public void draw(Graphics g) {
    g.setColor(this.color);
    switch (shape) {
      case CIRCLE:
        g.fillOval(center.x - radius,
            center.y - radius, 2 * radius,
                                2 * radius);
        break;
      case TRIANGLE:
        Polygon triangle = new Polygon();
        triangle.addPoint(center.x, center.y-radius);
        triangle.addPoint(center.x+radius, center.y+radius);
        triangle.addPoint(center.x-radius, center.y+radius);
        g.fillPolygon(triangle);
        break;
      case SQUARE:
        Polygon square = new Polygon();
        square.addPoint(center.x-radius, center.y-radius);
        square.addPoint(center.x+radius, center.y-radius);
        square.addPoint(center.x+radius, center.y+radius);
        square.addPoint(center.x-radius, center.y+radius);
        g.fillPolygon(square);
        break;
```

```
        case PENTAGON:
          Polygon pentagon = new Polygon();
          pentagon.addPoint(center.x, center.y-radius);
          pentagon.addPoint(center.x+radius,
            center.y-(radius/5));
          pentagon.addPoint(center.x+(3*radius/5),
            center.y+radius);
          pentagon.addPoint(center.x-(3*radius/5),
            center.y+radius);
          pentagon.addPoint(center.x-radius,
            center.y-(radius/5));
          g.fillPolygon(pentagon);
        default:
          g.fillOval(center.x - 5, center.y - 5, 10, 10);
          break;
      }
    }
}
```

它使用的配套 Shape 枚举如代码清单 3-24 所示。

代码清单 3-24：Shape 类

```
public enum Shape {
  CIRCLE,
  TRIANGLE,
  SQUARE,
  PENTAGON;

  @Override
  public String toString() {
    if (this == CIRCLE) {
      return "Circle";
    } else if (this == TRIANGLE) {
      return "Triangle";
    } else if (this == SQUARE) {
      return "Square";
    } else if (this == PENTAGON) {
```

```
        return "Pentagon";
      } else {
        return "unknown";
      }
    }
    public static Shape fromString(String s) {
      for (Shape shape : Shape.values()) {
        if (shape.toString().equals(s)) {
          return shape;
        }
      }
      throw new IllegalArgumentException("Cannot make shape from:
      " + s);
    }
}
```

从 DrawingObject 类中可以看到，画在画布上的图像有一些简单的属性：形状类型、坐标原点、描述形状大小的半径和颜色属性。

现在看看代码清单 3-25 所示的形成 DrawingBoard 客户端的 Java Swing 窗口背后的客户端端点。

代码清单 3-25：DrawingClient 类

```
import java.io.*;
import java.net.*;
import javax.websocket.*;
import jwsp.chapter3.drawing.data.DrawingDecoder;
import jwsp.chapter3.drawing.data.DrawingEncoder;
import jwsp.chapter3.drawing.data.DrawingObject;

@ClientEndpoint(
  decoders = { DrawingDecoder.class },
  encoders = { DrawingEncoder.class }
)
public class DrawingClient {
```

```
    private Session session;
    private DrawingWindow window;

    public DrawingClient(DrawingWindow window) {
      this.window = window;
    }

    @OnOpen
    public void init(Session session) {
      this.session = session;
    }

    @OnMessage
    public void drawingChanged(DrawingObject drawingObject) {
      window.addDrawingObject(drawingObject);
    }

public void notifyServerDrawingChanged(DrawingObject
drawingObject) {
    try {
      this.session.getBasicRemote().sendObject(drawingObject);
    } catch (IOException ioe) {
      System.out.println("Error: IO " + ioe.getMessage());
        } catch (EncodeException ee) {
          System.out.println("Error encoding object: "
          + ee.getObject());
        }
    }

        @OnError
    public void handleError(Throwable thw) {
      if (thw instanceof DecodeException) {
        System.out.println("Error decoding incoming message: "
        + ((DecodeException)thw).getText());
      } else {
        System.out.println("Client WebSocket error: " +
                                          thw.getMessage());
      }
```

```
    }

    public static DrawingClient connect(DrawingWindow window,
    String path) {
      WebSocketContainer wsc =
      ContainerProvider.getWebSocketContainer();
      try {
        DrawingClient client = new DrawingClient(window);
        wsc.connectToServer(client, new URI(path));
        return client;
      } catch (IOException ioe) {
        System.out.println("Error Connecting: " +
        ioe.getMessage());
      } catch (DeploymentException de) {
        System.out.println("Error deploying: " + de.getMessage());
      } catch (URISyntaxException ue) {
        System.out.println("Bad path: " + path);
      }
      return null;
    }

    public void disconnect() {
      if (this.session != null) {
        try {
          this.session.close();
        } catch (IOException ioe) {
          System.out.println("Error closing the session: " + ioe);
        }
      }
    }
  }
```

首先，注意到创建客户端端点时必须传入一个DrawingWindow对象。DrawingWindow是一个Swing JFrame组件，包含了所有的用户界面控件，以及当运行DrawingBoard应用客户端时的可视画布。客户端端点被构造好后，把DrawingWindow引

用保存为一个私有实例变量供后续使用。这里我们能看到客户端端点将利用两个类去编码和解码消息：DrawingEncoder类和DrawingDecoder类。这些类分别知道如何将DrawingObject对象编码成WebSocket消息以及将WebSocket消息解码成DrawingObject对象。后续将简要地介绍这些转换类是如何实现的，但现在只需要知道在这个客户端端点中，是通过在@ClientEndpoint注解中配置使用它们的。此外，客户端端点还使用了两个WebSocket生命周期事件，第一，在它的init(Session session)方法上通过使用@OnOpen注解来拦截打开事件。当建立与服务器的WebSocket连接时，这个方法将传入的Session对象保存为私有实例变量供后续使用。客户端端点使用的第二个生命周期注解是drawingChanged(DrawingObject drawingObject)方法上的@OnMessage注解。因为这个WebSocket端点为DrawingObject配置了解码器，所以能够把DrawingObject对象作为drawingChanged方法的一个参数。因此，在这种情况下，当收到WebSocket消息时，会把WebSocket消息转换成DrawingObject对象，这个方法会被调用。在drawingChanged()方法实现中我们看到，作为接收服务器端DrawingObject消息的响应，客户端端点会调用DrawingWindow绘制收到的对象。我们看到该客户端端点还通过@OnError注解声明了错误处理方法。

接下来看到DrawingEndpoint提供了notifyServerDrawingChanged(DrawingObject drawingObject)方法，它通过调用RemoteEndpoint的sendObject()方法，给DrawingBoard应用的服务器端发送了一个DrawingObject实例。在这背后，WebSocket实现会使用在类级别@ClientEndpoint注解中提供的编码器，也就是DrawingEncoder实例。为了把DrawingObject实例转换成WebSocket消息，在运行时会调用DrawingEncoder的encode方法。

注意错误处理方法 handleError()，它在 DrawingClient 中显式

地处理在解码入站消息时产生的各种错误。当然，在真正的应用中，这样的错误处理只是简单地打印输出错误信息，但handleError()方法会告诉你在代码中如何区分这些错误。

最后注意到DrawingClient有一个连接服务器的静态方法connect()。该方法实现的核心是WebSocketContainer类的connectToServer()方法，WebSocketContainer类用于将客户端端点的实例发布到提供的URL上。

接下来看看 DrawingWindow 本身，它是使用 DrawingClien 端点作为其模型的用户界面。我们不会花太多时间去深入介绍用户界面是如何工作的，因为本书是关于 Java WebSocket 的，而不是关于 Java Swing API 的。但我们还是注意到 DrawingWindow 中有如下比较有趣的事情：

- 当绘画窗口打开时，会调用静态 connect()方法初始化DrawingClient类与应用服务器端之间的连接。
- 每当用户在DrawingWindow的画布中画一个新对象时，DrawingWindow 会用一个 DrawingObject 实例调用notifyServerDrawingChanged()方法来要求DrawingClient端点通知服务器。
- 当DrawingWindow关闭时，会调用DrawingClient的disconnect()方法，结束与服务器的会话。

这些就是 DrawingWindow 和 DrawingClient 端点联系的关键点，保持了团体绘画工作的同步。这里不准备列出DrawingWindow类的代码，因为其主要是一些用户界面的代码。

稍作停留介绍一下 DrawingClient 使用的 DrawingEncoder 和DrawingDecoder 类。

首先是 DrawingEncoder，它负责把绘画对象转换成WebSocket 消息。如代码清单 3-26 所示。

代码清单 3-26：DrawingEncoder 类

```
import javax.websocket.EncodeException;
import javax.websocket.Encoder;
import javax.websocket.EndpointConfig;

public class DrawingEncoder implements
Encoder.Text<DrawingObject> {

  @Override
  public void init(EndpointConfig config) {}

  @Override
  public void destroy() {}

  @Override
  public String encode(DrawingObject drawingObject)
                              throws EncodeException {
    StringBuilder sb = new StringBuilder();
    sb.append(DrawingObject.MESSAGE_NAME);
    sb.append(",");
    sb.append(drawingObject.getShape().toString());
    sb.append(",");
    sb.append(drawingObject.getCenter().x);
    sb.append(",");
    sb.append(drawingObject.getCenter().y);
    sb.append(",");
    sb.append(drawingObject.getRadius());
    sb.append(",");
    sb.append(drawingObject.getColor().getRGB());
    return sb.toString();
  }
}
```

注意到该 DrawingEncoder 实现了 Encoder.Text<DrawingObject> 接口。这就告诉你它是用来编码 DrawingObject 的，把通过接口参数传入的DrawingObject对象编码成一个文本格式的 WebSocket

消息。

提示：

为了快速判定Java WebSocket Encoder或Decoder执行什么样的转换，只需要关注它实现的Encoder或Decoder接口。

我们看到 Encoder 接口定义了一些它自己的生命周期。当每个编码器实例在准备服务和完成服务时，都会调用init(EndpointConfig config)和 destroy()方法。如果你实现的编码器需要初始化或者清理很昂贵的资源时，这些方法就很有用。实现中最核心的地方是 encode()方法，它把绘画对象转换成一个字符串(把 DrawingObject 的所有属性值用逗号分隔拼接出来的字符串)。

在本章最后我们将看到，就像端点自身一样，WebSocket 实现会为每个对等连接创建一个编码器实例。因此，在我们的 DrawingBoard 应用中，因为每个 DrawingWindow 都连接到了 DrawingServer 端点，所以对于每个 DrawingWindow 都会有一个 DrawingEncoder 实例。在这个 DrawingEncoder 中，生命周期方法的实现是空的，因为不需要任何资源。

现在看看转换操作。当编码DrawingObject时，DrawingEncoder会把它转换成WebSocket文本消息发送给服务器。当服务器发送出关于新DrawingObject(已经添加到团体画布上的)的更新时，会把WebSocket文本消息发送给所有已连接的客户端。每个客户端都会使用DrawingDecoder把包含DrawingObject数据的文本消息转换成DrawingObject实例。如代码清单 3-27 所示。

代码清单 3-27：DrawingDecoder 类

```
import java.awt.Color;
import java.awt.Point;
import java.util.StringTokenizer;
import javax.websocket.Decoder;
import javax.websocket.EndpointConfig;
import javax.websocket.DecodeException;

public class DrawingDecoder implements
Decoder.Text<DrawingObject> {

  @Override
  public void init(EndpointConfig config) {}

  @Override
  public void destroy() {}

  @Override
  public DrawingObject decode(String s) throws DecodeException {
    StringTokenizer st = new StringTokenizer(s, ",");
    String message_name = st.nextToken();
    Shape shape = Shape.fromString(st.nextToken());
    Point center = new Point(new Integer(st.nextToken()),
                             new Integer(st.nextToken()));
    int radius = new Integer(st.nextToken());
    Color color = new Color(new Integer(st.nextToken()));
    return new DrawingObject(shape, center, radius, color);
  }

  @Override
  public boolean willDecode(String s) {
    return s.startsWith(DrawingObject.MESSAGE_NAME);
  }
}
```

同样，你能看到该DrawingDecoder不需要利用Decoder接口的

init()和destroy()生命周期方法。紧接着也看到因为DrawingDecoder实现了Decoder.Text<DrawingObject>接口，所以它必须能够完成WebSocket文本消息到DrawingObject实例的转换，确切地说就是与DrawingEncoder相反的处理过程。willDecode()方法负责对文本消息作初步的检查，判断这些消息自己是否能解码。如果willDecode()方法没有返回true，WebSocket实现就不会调用decode()方法，消息也不会以DrawingObject形式被传递。然而，如果willDecode()检查通过，WebSocket实现将会调用decode()方法，传递文本消息。你能看到DrawingDecoder中的decode()实现以逗号为分隔符，把文本消息拆解成绘画对象属性，并以这些属性重构出DrawingObject实例。这和DrawingEncoder的处理过程完全相反。

到此为止，我们几乎理解了该应用的全部。

DrawingServer 端点

接下来看看服务器端。通过@ServerEndpoint马上知道DrawingServer类是注解式WebSocket端点，映射URI是/draw，该URI是相对于WAR文件上下文根的。如代码清单 3-28 所示。

代码清单 3-28：DrawingServer 类

```
import java.awt.Color;
import java.io.IOException;
import java.io.Reader;
import java.util.Iterator;
import javax.websocket.*;
import javax.websocket.server.ServerEndpoint;
import jwsp.chapter3.drawing.data.DrawingAsIterator;
import jwsp.chapter3.drawing.data.DrawingFromReader;
import jwsp.chapter3.drawing.data.DrawingObject;

ServerEndpoint("/draw")
```

```
public class DrawingServer {
  private Session session;

  @OnOpen
  public void initSession(Session session) {
    this.session = session;
  }

  @OnMessage
  public void shapeCreated(Reader reader) {
    DrawingObject drawingObject;
    try (Reader rdr = reader) {
      DrawingFromReader dp = new DrawingFromReader(rdr);
      drawingObject = dp.getDrawingObject();

    } catch (IOException e) {
      System.out.println("There was an error reading the
      incoming message.");

      return;
    }
    DrawingObject toSend = new DrawingObject(
          drawingObject.getShape(),
          drawingObject.getCenter(),
          drawingObject.getRadius(),
        this.getFadedColor(drawingObject.getColor()));
    for (Session otherSession : this.session.getOpenSessions()) {
      if (!otherSession.equals(this.session)) {
        try {
          DrawingAsIterator dai = new DrawingAsIterator(toSend);
          sendDrawing(otherSession, dai);
        } catch (IOException ioe) {
          System.out.println("Communication error: " +
                                    ioe.getMessage());
        }
      }
    }
  }
```

```
    private Color getFadedColor(Color c) {
      Color faded = new Color((int) 255 - ((255-c.getRed()) / 2),
                              (int) 255 - ((255-c.getGreen()) / 2),
                              (int) 255 - ((255-c.getBlue()) / 2));
      return faded;
    }

    @OnError
    public void handleError(Throwable thw) {
      if (thw instanceof DecodeException) {
        System.out.println("Error decoding incoming message: " +
                           ((DecodeException)thw).getText());
      } else {
        System.out.println("Server WebSocket error: " +
                                          thw.getMessage());
      }
    }

    private void sendDrawing(Session aSession,
              Iterator<String> drawingAsIterator)
                    throws IOException {
      RemoteEndpoint.Basic remote = aSession.getBasicRemote();
      while (drawingAsIterator.hasNext()) {
        String partialMessage = drawingAsIterator.next();
        boolean isLast = !drawingAsIterator.hasNext();
        remote.sendText(partialMessage, isLast);
      }
    }
}
```

DrawingServer的目的是处理从已连接客户端发来的消息，这些消息代表要往画布中增加的DrawingObject对象。客户端每次发送这样的消息(被DrawingEncoder编码的DrawingObject对象)时，DrawingServer端点都会在shapeCreated()方法中处理它。你将注意到DrawingServer并没有使用任何Java WebSocket编码器或者解码

器去处理消息。相反，为了说明在本章前面提到的消息处理的其他方式，我们选择的消息处理形式是java.io.Reader，这是shapeCreated()方法的参数类型，它致力于以文本消息片段序列的形式发送消息。你会注意到shapeCreated()的实现使用了一个支持类DrawingFromReader，从Reader读取数据并构造DrawingObject实例。它把DrawingObject作为字符流处理，代码清单 3-29 如下所示。

代码清单 3-29：DrawingFromReader 类

```
import java.awt.Color;
import java.awt.Point;
import java.io.*;
import java.util.*;

public class DrawingFromReader {
    private Reader reader;

    public DrawingFromReader(Reader r) throws IOException {
      this.reader = r;
    }

  public DrawingObject getDrawingObject() {
    Iterator<String> readerItr = new
          DrawingObjectReaderIterator(this.reader);
    String message_name = readerItr.next();
    Shape shape = Shape.fromString(readerItr.next());
    Point center = new Point(new Integer(readerItr.next()),
          new Integer(readerItr.next()));
    int radius = new Integer(readerItr.next());
    Color color = new Color(new Integer(readerItr.next()));
    return new DrawingObject(shape, center, radius, color);
  }

  class DrawingObjectReaderIterator implements
```

```
Iterator<String> {
    private Reader reader;
    private boolean hasNext;

    DrawingObjectReaderIterator(Reader reader) {
      this.reader = reader;
    }

    @Override
    public boolean hasNext() {
      return hasNext;
    }

    @Override
    public String next() {
      try {
        StringBuilder sb = new StringBuilder();
        int i = 0;
        while ( (i=reader.read()) != -1 ) {
          if ((char) i == ',') {
            break;
          } else {
            sb.append((char) i);
          }
        }
        if (i == -1) {
          this.hasNext = false;
        }
        String s = sb.toString();
        return s;
      } catch (IOException ioe) {
        throw new RuntimeException("Error parsing from
        reader");
      }
    }

    @Override
    public void remove() {
```

```
            throw new UnsupportedOperationException();
        }
    }
}
```

然后 DrawingServer 的 shapeCreated()方法使用了 Session 对象经常用来广播消息的方法：

```
public Set<Session> getOpenSessions()
```

该方法返回所有已打开的会话集合。作为一个提醒，每个连接到该服务器端点的客户端都通过一个唯一的WebSocket连接来保持联系。因此，这个方法是列出所有已连接客户端的一个非常方便的方式，它调整了图形的颜色属性并给每个已连接的客户端(除了发送更新的客户端之外)广播新图形。更新每个客户端的操作委托给了sendDrawing()方法。该方法使用了一个支持类DrawingAsIterator，它能把DrawingObject转换成代表DrawingObject属性的字符串对象的迭代器。DrawingAsIterator类为转换使用了与DrawingEncoder相同的表示法，如代码清单 3-30 所示。

代码清单 3-30：DrawingAsIterator 类

```
import java.util.Iterator;

public class DrawingAsIterator implements Iterator<String> {
  private DrawingObject drawingObject;
  private int index = 0;

  public DrawingAsIterator(DrawingObject drawingObject) {
    this.drawingObject = drawingObject;
  }

  @Override
  public String next() {
```

```
        index++;
        switch (index) {
        case 1:
          return DrawingObject.MESSAGE_NAME;
        case 2:
          return ",";
        case 3:
          return drawingObject.getShape().toString();
        case 4:
          return ",";
        case 5:
          return Integer.toString(drawingObject.getCenter().x);
        case 6:
          return ",";
        case 7:
          return Integer.toString(drawingObject.getCenter().y);
        case 8:
          return ",";
        case 9:
          return Integer.toString(drawingObject.getRadius());
        case 10:
          return ",";
        case 11:
          return Integer.toString(drawingObject.getColor().
          getRGB());
        default:
          throw new IllegalStateException("No more elements");
        }
      }

      @Override
      public boolean hasNext() {
        return this.index < 11;
      }

      @Override
      public void remove() {
        throw new UnsupportedOperationException();
```

```
    }
}
```

DrawingServer 的 sendDrawing()方法使用了 Java WebSocket API RemoteEndpoint.Basic 的下列方法

```
public void sendText(String partialString, boolean isLast)
throws IOException
```

给每个已连接的客户端发送 DrawingObject 字符串表示形式的每个片段。在这种方式下，DrawingServer 使得所有的客户端及时更新团队画布上的内容。

让我们回过头来想想在这个示例中看到了什么。在客户端方面，你能看到我们利用了 Java WebSocket API 的编码器和解码器机制去处理 WebSocket 消息格式。这意味着 DrawingClient 端点只需要处理 DrawingObject 实例，不需要知道 DrawingObject 的 WebSocket 消息形式。这个工作是委托给编码器和解码器实现的。

在服务器端方面，DrawingServer 负责侦听任何已连接客户端上的变化并广播更新的工作，还负责把 DrawingObject 处理成基本 WebSocket 消息形式(在传入端，它必须解析成字符流；在传出端，它是字符串集合并以消息片段序列的形式发送回客户端)。

尽管 DrawingServer 端点以字符串片段序列的形式发送绘画更新内容，但是作为一次绘画更新，客户端需要完整的消息，因此 WebSocket 实现会缓冲入站消息直到所有片段被收到。只有这样做，DrawingDecoder 才能把它转换成描述绘画更新内容的 DrawingObject 对象。

为了说明在本章开头列出的一些选项，在 DrawingBoard 应用中，我们包含了一系列的消息处理的选择。在真实世界的应用中，你很可能只选择其中一种或者两种可选项。事实上，为了代码的

简洁，即使在你读完本章之前，你都可以尝试着使用 DrawingEncoder 和 DrawingDecoder 重构 DrawingServer 端点。

3.3 消息通信和线程

在结束本章前，我们转向一个对 Java 开发人员来说非常重要的话题：线程。很容易看到在绘画示例中当客户端数量增加时，就增加了两个客户端同时更新服务器的可能性。这也会增加一个客户端发送更新的同时服务器也正在更新其他客户端的可能性。作为 WebSocket 开发人员，对于有多少个线程会同时访问你的端点你该保证什么？特别是如果一个 WebSocket 消息正在以消息片段序列的形式发送，消息要按发送的顺序到达并且不能与其他消息混合，又该如何保证？

3.3.1 WebSocket 端点线程和消息通信

幸运的是，对于开发人员来说，Java WebSocket API 提供了大量简单的保证来回答这些问题：

- 每当有新的客户端连接进来时，Java WebSocket API 会创建一个新的服务器端点实例。
- 每个 WebSocket 端点实例在任意时刻都只能被一个线程调用。
- 当 WebSocket 消息以片段形式到达时，Java WebSocket API 必须确保以正确的顺序调用相应的端点，并且消息部分不会与其他消息交织。

上述 3 点为端点保证了一个非常明确的单线程模型，排除了之前描述的一些不利情形。这也意味着每个端点实例只能看到同

一个会话实例，该会话实例代表了那个唯一导致该端点实例被创建的客户端连接。会话以实例变量的形式保存在 DrawingServer 端点中。

3.3.2　线程与编码器和解码器的生命周期

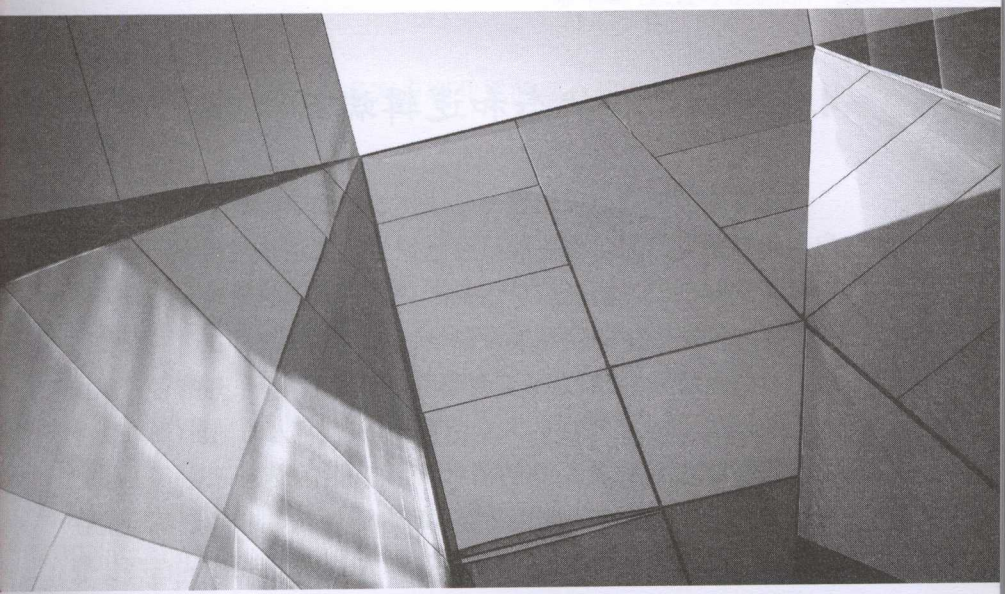
对于开发 Encoder 和 Decoder 类，Java WebSocket API 保证会为每一个 WebSocket 连接创建一个 Encoder 或者 Decoder 类的实例。该保证也适用于消息处理过程中，这就意味着，假如解码器的开发人员能够预料到同一时刻最多只会有一个线程访问 decode()方法，则无须开发代码来应对并发访问。连接建立后，在第一条需要它处理的消息到达时，会调用 Encoder 和 Decoder 实例的 init()方法，当连接关闭时调用 destroy()方法。

突然想到的一个问题是，特别是对于 WebSocket 应用的服务器端方面的开发，供所有客户端连接共用的应用状态会被存放在哪里？我们可能要扩展 DrawingBoard 应用以保留多个客户端合作的绘画图纸，或者可能希望当新的客户端加入团队时能看到画纸当时的全貌。因为对于每个新客户端来说都有一个新的服务器端点实例，所以我们不能在那里保存团队绘画图纸。我们将在第 4 章再回到这个端点状态的话题，该章中会介绍 Session 对象更多的属性和用法。

3.4　本章小结

本章中进一步地介绍了通过 Java WebSocket API 接收 WebSocket 消息的所有方法，包括了 WebSocket 文本消息、二进制消息和 Pong 消息的各种 Java 对象表示形式，还包括如何使用

@OnMessage 注解以及何时实现各种 MessageHandler 接口。我们介绍了使用 RemoteEndpoint.Basic API 以同步模式发送多种多样的 WebSocket 消息，还介绍了 Java WebSocket API 内置的编码和解码方案。最后，我们通过一个 DrawingBoard 示例说明了这些概念。



第4章

配置与 Session

本章将介绍 Java WebSocket API 中两个非常重要的对象：Session 和 EndpointConfig。这两个对象处于运行中的 WebSocket 应用的内部，分别代表了端点的配置信息和有效端连接，这是开发人员构造应用状态的基础。我们以 Chat 示例开始，该示例是 Java WebSocket 服务器应用的一个简单的 JavaScript 客户端，利用了 Session 和 EndpointConfig 的一些特征，为更详细地阐述这类 API 的用法作了铺垫。

4.1 Session 状态和逻辑端点状态

在本章之前我们讨论过一个团体绘画的示例，应用的多个用户合作画图。然而，你可能已经意识到，在这个应用中，团体绘画并不实际存在，只是用户连接成功后一系列的绘画更新。大多数重要应用所产生的信息或者影响比应用本身的存在时间要长，包括多用户协作的绘画(可重复给其他人看)、一系列的客人意见(一段时间后形成了餐厅的大众点评)、大量客人的地址(形成了一个披萨投递地图)、一系列的消息(一次对话的交谈记录)等，这把我们引到了如下两个问题上：在 WebSocket 应用中，在哪里能存放所有已连接客户端共有的应用状态？在绘图应用中，在哪里能存放集体绘画？

现在，Java 企业版(Java EE)开发人员有很多用于存放应用状态的选择，但这些选择都依赖于对从 Java WebSocket 应用中拉取应用状态的位置的理解。

在多客户端、单服务器的模式下，一定会存在以下两类应用状态：与单个连接端相关联的状态和与所有连接端相关联的状态。想象一个披萨投递应用，在应用中顾客可以输入他的订单以及地址。披萨投递团队定期从应用中获取订单列表和订单的投递地图。每个订单和地址是与单个客户端关联的应用状态，而订单列表和投递地图则是与多个客户端关联的应用状态。想象团体绘画应用，每个绘画图形是与单个客户端(绘画了该图形的客户端)关联的应用状态，而最终的图像则是与所有参与协作的客户端关联的应用状态。

本章的 Chat 示例是阐明这两类应用状态的一个非常好的方式，将直接引导我们进入这两个对象：Session(代表了单客户端的

状态)和 EndpointConfig(与多客户端共享的端点状态相关联)。

4.2　Chat 示例

Chat 应用会有很多变体：有的只允许群体聊天，聊天信息对所有人都是可见的；有的 Chat 应用列出了所有可聊天的对象，但只允许一对一的私聊；还有很多 Chat 应用既允许群聊也允许私聊。

我们的 Chat 示例属于群体聊天，有一个公共聊天副本存储了所有已登录用户的聊天信息。这是在聊天应用运行过程中建立的应用状态，对所有连接的用户是共有的。在这个应用中与单个已连接用户关联的应用状态是用户名。该 Chat 示例使用了 JavaScript 客户端连接 WebSocket 服务器端点。在此示例中，包括本书余下的部分示例，在列出代码清单时我们将限制为只列出与学习 Java WebSocket API 最相关的部分。

开始前看一下 Chat 示例运行后的截图，首先是用户登录(如图 4-1 所示)，接着是在几个用户登录后，聊天室变得活跃，形成了长长的聊天记录(如图 4-2 所示)。

在看代码之前，先快速了解一下消息交互过程，如图 4-3 所示。在客户端和服务器之间的 WebSocket 连接确定后，从图中可以看出聊天交谈以客户端发送登录请求消息开始，该消息包含了一个客户端希望使用的用户名。Chat 服务器立即作出响应，响应消息是客户端在聊天期间可以使用的已确认过的名字，这就使得服务器能够为每个加入聊天交谈的客户端选择一个独一无二的名字。很明显，如果 Chat 应用允许两个客户端使用相同的名字，会显得很混乱！

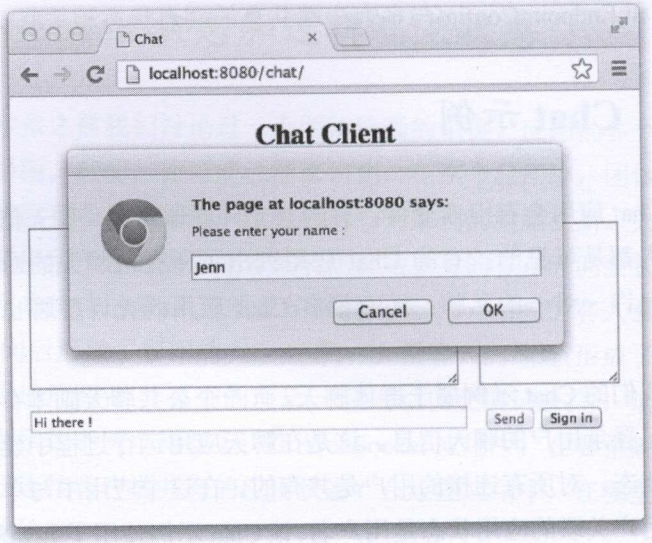


图 4-1　Chat 示例登录窗口

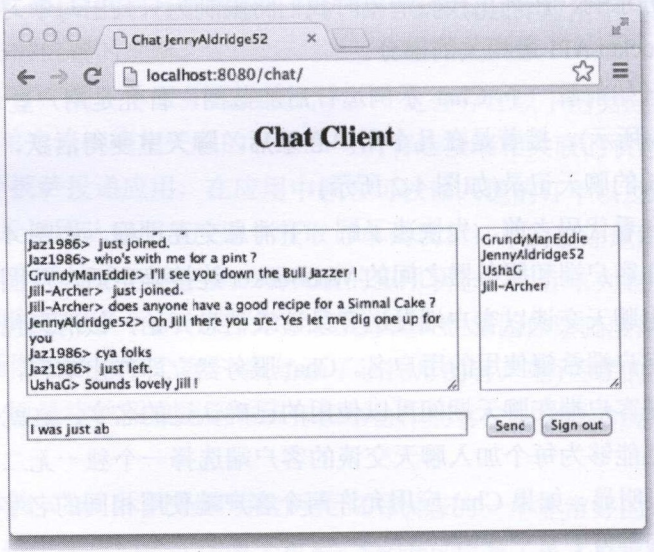


图 4-2　聊天窗口

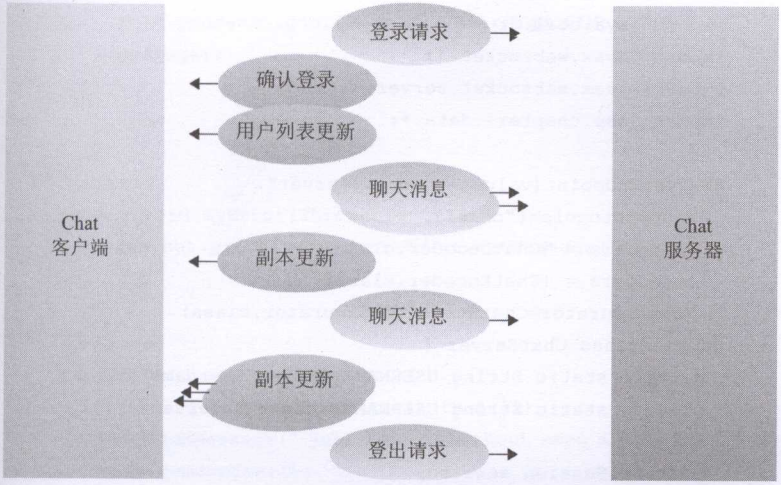


图 4-3　聊天消息交互图

每当新用户登录时，Chat 服务器发布一个消息给所有客户端，消息包含 Chat 示例里所有已登录的当前用户列表。每当用户在聊天窗口里输入一条新消息时，Chat 客户端给服务器发送聊天消息。服务器把收到的聊天消息与发送方用户名一起增加到聊天记录(这里存放了所有客户端发送过的聊天信息)中，然后会发布一个聊天记录更新的消息给所有已连接的客户端，以便这些客户端能够显示刚刚添加的新消息。当用户结束聊天时，登出过程是 Chat 客户端发送一个登出请求给服务器，服务器更新剩余的用户列表，同时发送一个副本消息告知用户已经退出，然后关闭 WebSocket 连接。

既然我们已经理解了 Chat 示例的消息交互过程，就来看看 Chat 示例的核心——ChatServer 端点，代码如代码清单 4-1 所示。

代码清单 4-1：ChatServer 端点

```
import java.io.IOException;
```

```
import java.util.*;
import javax.websocket.*;
import javax.websocket.server.*;
import jwsp.chapter4.data.*;

@ServerEndpoint(value = "/chat-server",
    subprotocols={"chat"},
    decoders = {ChatDecoder.class},
    encoders = {ChatEncoder.class},
    configurator=ChatServerConfigurator.class)
public class ChatServer {
  private static String USERNAME_KEY = "username";
  private static String USERNAMES_KEY = "usernames";

  private Session session;
  private ServerEndpointConfig endpointConfig;
  private Transcript transcript;

@OnOpen
public void startChatChannel(EndpointConfig config,
Session session) {
  this.endpointConfig = (ServerEndpointConfig) config;
  ChatServerConfigurator csc = (ChatServerConfigurator)
  endpointConfig.getConfigurator();
  this.transcript = csc.getTranscript();
  this.session = session;
}

@OnMessage
public void handleChatMessage(ChatMessage message) {
  switch (message.getType()){
    case NewUserMessage.USERNAME_MESSAGE:
      this.processNewUser((NewUserMessage) message);
      break;
    case ChatMessage.CHAT_DATA_MESSAGE:
      this.processChatUpdate((ChatUpdateMessage) message);
      break;
    case ChatMessage.SIGNOFF_REQUEST:
```

```
        this.processSignoffRequest((UserSignoffMessage)
        message);
    }
  }

  @OnError
  public void myError(Throwable t) {
    System.out.println("Error: " + t.getMessage());
  }

  @OnClose
  public void endChatChannel() {
    if (this.getCurrentUsername() != null) {
      this.addMessage(" just left...without even signing out !");
      this.removeUser();
    }
  }

  void processNewUser(NewUserMessage message) {
    String newUsername =
    this.validateUsername(message.getUsername());
    NewUserMessage uMessage = new NewUserMessage(newUsername);
    try {
      session.getBasicRemote().sendObject(uMessage);
    } catch (IOException | EncodeException ioe) {
      System.out.println("Error signing " + message.getUsername()
          + " into chat : " + ioe.getMessage());
    }
    this.registerUser(newUsername);
    this.broadcastUserListUpdate();
    this.addMessage(" just joined.");
  }

  void processChatUpdate(ChatUpdateMessage message) {
    this.addMessage(message.getMessage());
  }

  void processSignoffRequest(UserSignoffMessage drm) {
```

```
    this.addMessage(" just left.");
    this.removeUser();
  }

  private String getCurrentUsername() {
    return (String)
    session.getUserProperties().get(USERNAME_KEY);
  }

  private void registerUser(String username) {
    session.getUserProperties().put(USERNAME_KEY, username);
    this.updateUserList();
  }

  private void updateUserList() {
    List<String> usernames = new ArrayList<>();
    for (Session s : session.getOpenSessions()) {
      String uname = (String)
      s.getUserProperties().get(USERNAME_KEY);
      usernames.add(uname);
    }
    this.endpointConfig.getUserProperties().put(USERNAMES_KEY,
    usernames);
  }

  private List<String> getUserList() {
    List<String> userList = (List<String>)
    this.endpointConfig.getUserProperties().
      get(USERNAMES_KEY);
    return (userList == null) ? new ArrayList<String>() : userList;
  }

  private String validateUsername(String newUsername) {
    if (this.getUserList().contains(newUsername)) {
      return this.validateUsername(newUsername + "1");
    }
    return newUsername;
  }
```

```
private void broadcastUserListUpdate() {
  UserListUpdateMessage ulum =
    new UserListUpdateMessage(this.getUserList());
  for (Session nextSession : session.getOpenSessions()) {
      try {
        nextSession.getBasicRemote().sendObject(ulum);
    } catch (IOException | EncodeException ex) {
        System.out.println("Error updating a client : "
          + ex.getMessage());
    }
  }
}

private void removeUser() {
  try {
    this.updateUserList();
    this.broadcastUserListUpdate();
    this.session.getUserProperties().remove(USERNAME_KEY);
    this.session.close(new
      CloseReason( CloseReason.CloseCodes.NORMAL_CLOSURE,
      "User logged off"));
    } catch (IOException e) {
      System.out.println("Error removing user");
    }
  }

  private void broadcastTranscriptUpdate() {
    for (Session nextSession : session.getOpenSessions()) {
      ChatUpdateMessage cdm = new ChatUpdateMessage(
        this.transcript.getLastUsername(),
        this.transcript.getLastMessage());
      try {
        nextSession.getBasicRemote().sendObject(cdm);
      } catch (IOException | EncodeException ex) {
        System.out.println("Error updating a client : "
            + ex.getMessage());
      }
```

```
    }
  }

  private void addMessage(String message) {
    this.transcript.addEntry(this.getCurrentUsername(),
    message);
  this.broadcastTranscriptUpdate();
  }
}
```

让我们从学习它的实例变量持有的数据开始来查看这个端点，它持有一个 Session 实例的引用。你会记得，因为对于每个客户端连接都会有一个服务器端点的实例，所以 ChatServer 实例持有的这个引用就是与客户端连接相关的唯一 Session 实例。

ChatServer 要求这个 Session 实例传入它的所有生命周期方法中，这个 Session 也是@OnOpen、@OnMessage、@OnError 和@OnClose 注解方法的一个可选参数，但由于一些 ChatServer 应用逻辑被分解成多个单独的方法，与其在每个方法调用时都传递 Session 对象，不如把 Session 引用保存为实例变量来得方便。许多端点没有这个设计选择，在某些特殊的消息发送情形(即不是响应输入消息的消息发送)下，持有 Session 对象的引用就能访问 RemoteEndpoint 以发送消息。ChatServer 使用的第二个数据片是 ServerEndpointConfig 对象。你可能记得在第 1 章中部署编程式端点时创建过 ServerEndpointConfig，ServerEndpointConfig 对象持有配置 WebSocket 端点需要的所有配置数据和算法。我们将简单地讨论一下它们的细节，但现在我们注意到包含了所有配置信息的 ServerEndpointConfig 被声明在 ChatServer 端点使用的类级别@ServerEndpoint 注解中。还有个非常有用的特性就是 WebSocket 实现为所有 ChatServer 端点实例只会精确地创建一个 ServerEndpointConfig。这意味着 ServerEndpointConfig 是一个非

常有用的地方，可以存放所有 ChatServer 端点客户端共有的数据。换句话说，它非常适合存放当前已登录用户的列表，也适合存放聊天记录。基于 ServerEndpointConfig 的这个特性，在 Chat 应用中使用了两种不同的途径存放全局应用状态数据片。第一，你能看到当前已登录用户列表存放在 ServerEndpointConfig 的一个属性 Map 中；第二，从@ServerEndpoint 注解定义中能注意到在配置器属性中定义了一个 ChatServerConfigurator 类，这个类持有了其他的全局应用状态数据片——聊天记录。让我们看看代码清单 4-2 所示的配置器类：

代码清单 4-2：ChatServerConfigurator 类

```
import javax.websocket.HandshakeResponse;
import javax.websocket.server.HandshakeRequest;
import javax.websocket.server.ServerEndpointConfig;

public class ChatServerConfigurator
extends ServerEndpointConfig.Configurator {
 private Transcript transcript;

 public ChatServerConfigurator() {
  this.transcript = new Transcript(20);
 }

 public Transcript getTranscript() {
  return this.transcript;
 }

 @Override
 public void modifyHandshake(ServerEndpointConfig sec,
         HandshakeRequest request,
         HandshakeResponse response) {
System.out.println("Handshake Request headers: "
    + request.getHeaders());
```

```
        System.out.println("Handshake Response headers: "
            + response.getHeaders());
    }
}
```

在重写了 ServerEndpointConfig.Configurator 之后，这个类能够拦截来自 WebSocket 实现的配置回调。为了能简单地记录 HTTP 请求和响应的 Header 日志，这个类选择拦截打开阶段握手的 HTTP 请求和响应消息。该类也存放了我们之前提到过的聊天记录，聊天记录用 Transcript 类定义，代码如代码清单 4-3 所示。

代码清单 4-3：Transcript 类

```
import java.util.ArrayList;
import java.util.List;

public class Transcript {
  private List<String> messages = new ArrayList<>();
  private List<String> usernames = new ArrayList<>();
  private int maxLines;

  Transcript(int maxLines) {
    this.maxLines = maxLines;
  }

  public String getLastUsername() {
    return usernames.get(usernames.size() -1);
  }

  public String getLastMessage() {
    return messages.get(messages.size() -1);
  }

  public void addEntry(String username, String message) {
    if (usernames.size() > maxLines) {
        usernames.remove(0);
```

```
            messages.remove(0);
        }
        usernames.add(username);
        messages.add(message);
    }
}
```

因为 WebSocket 实现为所有的 ChatServer 端点实例只实例化一个 ServerEndpointConfig 实例，所以 ServerEndpointConfig 配置器引用的单个聊天记录就是被所有 ChatServer 端点实例共享的聊天记录。

回到 ChatServer 的代码，ChatServer 通过 ServerEndpointConfig 持有了聊天记录的引用，这就确保了所有用户的聊天消息被写入同一个全局聊天记录中。

注意到当 WebSocket 连接确立时这些实例变量才会被准备好，确立连接的是标有@OnOpen 注解的 startChatChannel()方法。

还要注意，ChatServer 从不直接处理基本的文本或者二进制格式的 WebSocket 消息。取而代之的是，它配置了 Encoder 实现和 Decoder 实现，意味着它只能发送 ChatMessage 实例的消息，也只能接收 ChatMessage 形式的消息。处理入站消息的方法是注解了@OnMessage 的 handleChatMessage()方法。

能看到当前已登录用户的用户名信息存放在 Session 对象的属性 Map 中。注意到在应用中有两种方法供用户退出交谈：第一个是友好的方法，另外一个是粗鲁的方法。如果用户点击了登出按钮，客户端会按照我们之前提到过的方式，发送一个登出请求给服务器。服务器相应地从当前会话中清除该用户名，并向所有其他已连接客户端广播用户列表的变化和附加的告别消息，然后才关闭 WebSocket 连接。然而，如果用户直接离开了客户端页面或者关闭了浏览器窗口，则客户端代码简单地关闭 WebSocket 连

接。在这种情况下，ChatServer 端点在 endChatConversation()方法(使用@OnClose 注解)被调用时会凭经验认为他已退出了聊天交谈。此时，它会删除用户并以不太友好的消息通知其他已登录用户。

也能注意到在@ServerEndpoint 注解中有个子协议属性，属性值是一个列表，包含一个元素"chat"。WebSocket 子协议是一个非常轻量级的方式，定义了端点希望使用一个特定的 WebSocket 消息格式和编排。在这个示例中，我们把聊天消息的特定集合和特定的发送顺序称为 chat 子协议，并且已经声明 ChatServer 端点使用该子协议。你也将注意到 JavaScript 客户端在创建它与 EchoServer 的 WebSocket 连接时也声明了该子协议。我们这里所说的 chat 协议对 Chat 应用是完全具体的，它是一个简单有用的标记方式，标记了应用的客户端和服务器都认同的公共消息格式和编排。稍后我们将学习更多关于 WebSocket 子协议的知识。

从这个 Chat 示例中总结出的主要收获有如下几点。

- 它使用了自定义的编码器和解码器来发送和接收消息。
- 它把用户相关的状态(用户名)存放在与 ChatServer 实例相关联的 Session 对象中。
- 它用ServerEndpointConfig对象的两个不同属性(分别是它的属性Map和一个自定义ServerEndpointConfig.Configurator)存放所有用户公用的应用状态(已登录用户列表和聊天记录)。
- 它定义了一个名为 chat 的 WebSocket 子协议，用来标记它使用的特定消息格式和编排。

注意：

当设计在哪里存放应用中所有用户公用的应用状态时，可以选择使用 ServerEndpointConfig 的属性 Map，也可以自己实现

ServletEndpointConfig.Configurator，这两个途径都是有效的。一般来说，对于单一不变的对象，使用属性 Map 是一个轻量级方法，因为它不需要创建新类。但如果应用数据非常复杂，需要进行数据计算或者希望类型更安全，就需要考虑扩展 ServletEndpointConfig.Configurator。

查看这个应用余下的代码会感到很轻松。接下来，将相对正式地介绍 EndpointConfig 和 Session 对象(它们是本示例的核心)。

4.3　配置端点：ClientEndpointConfig 和 ServerEndpointConfig

在本节中，我们将详细讨论各种可以被 WebSocket 端点开发人员使用的配置选项，无论这些端点是部署在服务器上还是想作为客户端应用的一部分，也无论它是编程式端点还是注解式端点。

4.3.1　提供和访问端点配置信息

在 Java WebSocket API 中，使用 EndpointConfig 类里定义的信息和算法来配置 WebSocket 端点。服务器端端点的配置使用的是 ServerEndpointConfig 类提供的信息，客户端端点的配置使用的是 ClientEndpointConfig 类提供的信息。

如果只使用注解式端点，将会看到使用@ClientEndpoint 或@ServerEndpoint 注解可以提供配置信息。WebSocket 信息使用这些注解提供的信息来分别创建 ClientEndpointConfig 或 ServerEndpointConfig 实例，这些实例可以被用在部署时配置每个端点实例，也可作为生命周期方法的可选参数。

如果只使用编程式端点，将会知道通过使用ClientEndpointConfig.

Builder 类或者 ServerEndpointConfig.Builder 类来构造合适的 EndpointConfig实例，可以为你的端点提供配置信息。对于服务器端端点，需要提供在ServerApplicationConfig实现中想使用的 ServerEndpointConfig实例，例如在第 1 章中部署编程式Echo示例时。对于客户端端点，需要提供你想使用的ClientEndpointConfig 实例给关联的WebSocketContainer.connectToServer()方法，该方法用来部署端点。正如注解式端点一样，WebSocket实现用来构造端点实例的EndpointConfig对每个端点实例来说都是可用的，可通过 public abstract void onOpen(Session session, EndpointConfig config) 方法来使用，这是编程式Endpoint类的子类必须实现的方法。

注意：

在所有情况下，当端点部署后，对于所有端点实例来说，只有一个存放配置信息的 EndpointConfig 实例。

在我们更深入学习 EndpointConfig 类之前，先总结一下不同的配置，参考表 4-1。

表 4-1 提供和访问端点配置的总结

端点类型	配置信息的提供方式	EndpointConfig 的可用性
编程式服务器	在提供的 ServerApplicationConfig 中构造一个 ServerEndpointConfig	在Endpoint.onOpen()方法中
注解式服务器	在@ServerEndpoint 中声明	作为端点生命周期方法的一个可选参数
编程式客户端	在 WebSocketContainer.connectToServer() 方法中构造一个 ClientEndpointConfig	在Endpoint.onOpen()方法中
注解式客户端	在@ClientEndpoint 中声明	作为端点生命周期方法的一个可选参数

4.3.2　配置选项介绍

无论是客户端端点还是服务器端端点，所有端点配置的公有属性如下：

- 运行时可通过 EndpointConfig.getEncoders()方法获得一个编码器列表。
- 运行时可通过 EndpointConfig.getDecoders()方法获得一个解码器列表。
- 运行时可通过EndpointConfig.getUserProperties()获得一个开发人员对象的属性Map。

表 4-2 总结了客户端端点和服务器端端点的公有配置属性是如何被添加到端点定义中的。

表 4-2　在端点定义中添加公有配置属性的方法

配置属性	注解式端点	编程式端点
编码器列表	@ServerEndpoint 或 @ClientEndpoint 的 encoders()属性	ClientEndpointConfig.Builder.encoders (List<Class<? extends Encoder>> encoders) 或者 ServerEndpointConfig.Builder.encoders (List<Class<? extends Encoder>> encoders)
解码器列表	@ServerEndpoint 或 @ClientEndpoint 的 decoders()属性	ClientEndpointConfig.Builder.decoders(List<Class<? extends Decoder>> decoders) 或者 ServerEndpointConfig.Builder.decoders(List<Class<? extends Decoder>> decoders)
开发人员对象 Map	EndpointConfig. getUserProperties()	EndpointConfig.getUserProperties()

编码器列表是一组实现了 Encoder 接口的 Java 类，当发送对象时，会用这些编码器来编码开发人员对象。而对于那些标准类型(如 String、byte[]、ByteBuffer、Java 原生类型或其等价类)对象，Java WebSocket API 会自动处理。

在你的应用中，可能需要发送不同的 Java 对象给对方。在这种情况下，就需要提供多个编码器。此时，Java WebSocket 实现会从列表中选择第一个能对发送对象进行编码的编码器。例如，如果你有两个 Banana 类的编码器，一个实现了 Encoder.Text<Banana>接口，另一个实现了 Encoder.Binary<Banana>。你的应用是以文本形式还是以二进制形式发送 Banana 对象，取决于在编码器列表(在 WebSocket 注解中或者在 EndpointConfig 中提供)中实现了 Encoder.Text<Banana>的编码器是在实现了 Encoder.Binary<Banana>的编码器的前面还是后面。

解码器列表是一组实现了Decoder接口的Java类。当端点处理到达的WebSocket消息时，WebSocket实现会使用第一个特定的编码器，该编码器能够处理基本类型(文本或者二进制)消息，而且它的willDecode()方法返回true。例如，如果你的应用选择接收Banana对象，并且你已经配置了一些能够解码WebSocket消息成Banana实例的解码器类，如果到达的是二进制消息，则WebSocket实现会从解码器列表中寻找第一个能够把二进制WebSocket消息解码成Banana实例的解码器类(也就是说，实现Decoder.Binary<Banana>或者Decoder.BinaryStream<Banana>的类)。如果解码器是一个Decoder.Binary<Banana>，则Websocket实现会执行额外的检查，看解码器的willDecode()方法是否返回true，如果没有返回true，则从解码器列表中继续依次寻找。

开发人员对象 Map 是一个可编辑的 java.util.Map，键是String，值是对象，开发人员会用它来存储一些对所有端点实例来

说都很常见的应用状态。像在 Chat 示例中，开发人员对象 Map 是一个存放和保持当前已登录用户的好地方。

在查看各种客户端和服务器端的 WebSocket 端点配置属性之前，我们需要先理解一下建立 WebSocket 网络协议的一对机制。

4.3.3　WebSocket 子协议和 WebSocket 扩展

即使在一个相对简单的应用程序中(如 Chat 示例)，我们也看到其为文本消息定义了一个特殊的格式用于客户端和服务器通信，还定义了一个特殊的消息交互的时序或者编排。事实上，所有的 WebSocket 应用都会定义一些特定于具体应用的消息格式或编排。WebSocket 协议把这个格式或编排称为 WebSocket 子协议，子协议建立在原始 WebSocket 消息格式和连接生命周期之上。WebSocket 子协议用简单的字符串名称来标记。在 Chat 示例中，我们把聊天消息的特殊编排定义成 chat 子协议。很有可能会出现一些标准的 WebSocket 协议，这种情形下，在思考应用的子协议名时要非常小心。在 www.iana.org 上有个 WebSocket 子协议注册表，在那里能看到子协议名称通常被用来标记应用，通过一个简洁的名字来封装应用的特定会话约定。

在更深的层次，WebSocket 协议允许在数据帧(用于在端点间传递事件和数据)内插入任意数据。这些数据可以用于在承载连接的中间服务器间进行帧路由，也可以用于描述或限定数据帧内的 WebSocket 数据。如果 WebSocket 实现决定使用协议的这种能力，在这样做时，就需要扩展 WebSocket 协议，这种方案被称为 WebSocket 扩展。

WebSocket 扩展是配置扩展的一个字符串名称和一个名-值对集合。一小部分扩展已经被正式提出来，如名为 DEFLATE 的扩展允许在 WebSocket 消息中压缩数据以改进传输效率，名为 MUX

的扩展则允许多个逻辑的 WebSocket 连接运行在一个物理 TCP 连接之上以提高网络可伸缩性。随着 WebSocket 的流行，必定会出现大量的子协议和扩展。

1. 纯客户端配置属性

客户端端点能够指定如下配置属性：

- 在运行时通过ClientEndpointConfig.getPreferredSubprotocols()可获得一个WebSocket子协议名称的列表。
- 在运行时通过ClientEndpointConfig.getExtensions()可获得一个WebSocket扩展的列表。
- 在运行时通过 ClientEndpointConfig.getConfigurator()可获得 ClientEndpoint.Configurator 类。

WebSocket子协议名称列表是一个包含String对象的列表，是配置的一部分。该列表在开始握手期间使用，也就是说，当WebSocket与服务器建立连接时，会协商出一个双方均同意使用的子协议。扩展列表是一个javax.websocket.Extension对象列表，每个扩展都封装了描述WebSocket扩展的名称和扩展属性列表。通过这个属性可以指定在客户端端点连接到服务器时需要使用的所有扩展，以及这些扩展应用于WebSocket连接协议的顺序。像子协议一样，在WebSocket连接握手期间有一个协商过程，使得WebSocket连接的两端都同意这些WebSocket扩展，在建立连接时会使用这些扩展。在 WebSocket 实现中可通过 WebSocketContainer 的 getInstalledExtensions()方法获得WebSocket扩展列表。

当我们了解更多关于服务器端点配置的细节时，就更能理解协商 WebSocket 子协议和 WebSocket 扩展是如何可定制的。

ClientEndpointConfig.Configurator类可以用来拦截HTTP请求(底层WebSocket实现主动发出的消息)和HTTP响应(从WebSocket

服务器接收到的返回消息)。如果没有显式地提供该配置器类，WebSocket实现会提供一个默认的。该类有两个方法。第一个方法是：

```
public void beforeRequest(Map<String,List<String>> headers)
```

该方法在 WebSocket 实现发出握手请求(为了初始化连接需要发送该请求)后被调用。如果提供了自己的 Configurator 实现，则传入此方法的 headers 参数能被增加或者修改。第二个方法是：

```
public void afterResponse(HandshakeResponse hr)
```

该方法是 WebSocket 实现从 WebSocket 服务器收到了握手响应后被调用，可能会检查 HandshakeResponse 的头信息。

提供自定义 ClientEndpointConfig.Configurator 来配置客户端端点的最常见的使用场景是在握手请求中插入自定义 HTTP 头或者是修改它，如插入或更新 HTTP Cookie 或者认证信息，以及从服务器读取 Cookie 或者认证要求。这是 Java WebSocket API 更高级的使用，许多开发人员不需要使用它。

然而，继承 ClientEndpointConfig.Configurator 时添加一些特定于应用或者框架的功能是可能的，例如在端点中访问数据库或者远程 Web 服务。事实上，可以使用相同的 ClientEndpointConfig.Configurator 实例来配置多个不同的客户端端点，这样多个客户端端点能共享相同的服务，如数据库连接。这使得在你的客户端 WebSocket 应用中，通过配置器来引入其他服务或者功能是一种非常灵活的方式。

在本节最后我们总结一下纯客户端配置属性是如何添加到端点定义中的，参见表 4-3。

表 4-3 在端点定义中增加纯客户端配置属性的方法

配置属性	注解式端点	编程式端点
子协议	@ClientEndpoint 的 subprotocols()属性	ClientEndpointConfig.Builder .preferredSubprotocols(List<String> subprotocols)
扩展	@ClientEndpoint 的 extensions()属性	ClientEndpointConfig.Builder .extensions(List<Extension> extensions)
配置器	@ClientEndpoint 的 configurator()属性	ClientEndpointConfig.Builder .configurator(ClientEndpointConfig .Configurator configurator)

2. 纯服务器端配置属性

服务器端端点可以配置如下属性：

- 在运行时调用ServerEndpointConfig.getPath()可获得端点路径。
- 在运行时调用ServerEndpointConfig.getEndpointClass()可获得端点类。
- 在运行时调用ServerEndpointConfig.getSubprotocols()可获得端点支持的子协议列表。
- 在运行时调用ServerEndpointConfig.getExtensions()可获得端点支持的扩展列表。
- 在运行时调用 ServerEndpointConfig.getConfigurator()可获得 ServerEndpointConfig.Configurator。

这里的路径是相对于Web应用上下文根(WebSocket实现用来

存放端点)来说的相对路径。路径可以是一个 URI 字符串，这是目前为止我们在示例中使用的唯一形式。路径还可以是一个 URI 模板(级别 1)。URI 模板的复杂性有各种不同的级别，级别 1 类型的模板总是这样的形式：一个或者多个 URI 路径段作为变量，变量的表达式是{*varname*}，*varname* 是变量名。以下是一些合法的路径：

```
/chat
/travel/hotels
/member/{level}
/cars/{make}/{model}
```

我们将在第 6 章回到 Java WebSocket API 中的与路径映射相关的主题。

端点类属性把服务器端点类和它使用的配置联系起来，确保同一类端点使用的 ServerEndpointConfig 实例是相同的。

子协议属性定义了服务器端点支持的 WebSocket 子协议名称列表，以优先级从高到低排序。扩展属性定义了服务器端点在客户端连接它时会用到的 WebSocket 扩展列表，以优先级从高到低排序，优先级最高的排在列表第一位。

接下来，我们将讨论当客户端尝试发起连接时，客户端和服务器端点是如何决定使用哪个子协议和哪些扩展的。

与 ClientEndpointConfig.Configurator 很像的是，ServerEndpointConfig.Configurator 是一个定义了 WebSocket 服务器端是如何进行握手的类。如果你没有提供它，则 WebSocket 实现会提供一个默认的。然而，如果你提供了一个自定义 ServerEndpointConfig.Configurator，就能重写一些标准算法(属于握手的一部分)。让我们看一下：

```
public String getNegotiatedSubprotocol(List<String>
    supported,List<String> requested)
```

在握手过程中，当需要知道连接会使用哪些 WebSocket 子协议时，WebSocket 实现会调用该方法。第一个参数是服务器端点支持的子协议列表，第二个参数是发送握手请求时客户端支持的子协议列表。

如果不提供重写上述方法的自定义 ServerEndpointConfig.Configurator，则 WebSocket 实现会从客户端支持的列表中选择第一个在服务器端同样支持的子协议。如果两个列表不存在交集，则返回空字符串。例如，如果客户端请求的子协议是{'chat-53', 'chat', 'old-chat'}，端点支持的子协议是{'chat-12', 'old-chat', 'chat', 'quick-chat'}，该方法的默认实现会返回子协议 chat。这是因为，尽管服务器端点支持 old-chat，但是在请求列表中 chat 排在前面。

接下来的 ServerEndpointConfig.Configurator 方法是：

```
public List<Extension> getNegotiatedExtensions(
            List<Extension> installed,
            List<Extension> requested)
```

与上面提到的getNegotiatedSubprotocol()方法很像，在握手期间，会调用该方法来确定使用的WebSocket扩展。如果开发人员没有重写该方法，WebSocket实现会从它已安装的扩展列表和客户端请求的扩展列表中选择共有的，以请求的扩展列表顺序为准。扩展的返回顺序非常重要。假设你有两个WebSocket扩展，能够编码输出的WebSocket消息和解码传入的WebSocket消息。假设第一个扩展是压缩数据，第二个扩展是加密。很显然，以相反的顺序(相反于编码顺序)解码扩展非常重要。所以，端点的两端必须就WebSocket扩展的使用顺序达成一致，保证使用相

同的顺序。WebSocket实现使用已达成共识的扩展，顺序使用这个方法返回的扩展列表。如果想重写这个方法，不要忘记，正如在 ClientEndpointConfig.Configurator 中所见，可以通过WebSocketContainer.getInstalledExtensions()方法获得WebSocket实现所有已安装的扩展列表。

下一个在自定义 ServerEndpointConfig.Configurator 类中可以重写的方法是：

```
public boolean checkOrigin(String originHeaderValue)
```

所有的浏览器客户端在握手请求中都会携带 HTTP 源头，以便 WebSocket 实现能够判定 WebSocket 连接是否来源于页面(从 WebSocket 实现的 Web 服务器端加载的页面)。该方法封装了安全检查。大多数开发人员不需要重写它，但是当开发人员希望提供一个基于不寻常设置的更严格的安全检查时可以重写它。

每当 WebSocket 实现需要获取一个服务器端点实例时(因为有新的客户端连接进来)，都会调用如下方法：

```
public <T> T getEndpointInstance(Class<T> endpointClass)
        throws InstantiationException
```

该方法的默认实现会返回一个新的 WebSocket 端点实例，这就是为什么 WebSocket 实现能够精确地保证每个客户端连接都对应一个服务器端点的实例。然而，如果你希望控制服务器端点的实例数量，那么可以重写该方法。有些开发人员可能以这样的方式设计端点：只有一个服务器端点实例来处理所有客户端连接。在这种情况下，当实现服务器端点时，就要更多考虑多个并发的 WebSocket 生命周期事件。尽管 WebSocket 实现不允许同一个客户端同时有多个Java线程调用服务器端点，但如果有多个客户端，

则可能会有多个线程同时访问。如果你希望尝试这种高级的端点配置模式，就需要确保该方法在每次调用时都能返回相同的唯一的服务器端点实例。

最后，ServerEndpointConfig.Configurator最强大的配置方法是：

```
public void modifyHandshake(ServerEndpointConfig sec,
    HandshakeRequest request,
    HandshakeResponse response)
```

该方法能让你完整地读取客户端尝试连接时发出的握手请求，也能让你在WebSocket实现创建的握手响应返回给客户端之前完整地读/写它们。如果你学习了握手请求和响应的精确格式，就能知道握手响应中有一部分包括了协商好的子协议和扩展。正如之前我们看到过的，这些协商是调用ServerEndpointConfig.Configurator的其他方法的返回结果。等到modifyHandshake()方法被调用时，其他的方法(getNegotiatedSubprotocol()和getNegotiatedExtensions())就已经被调用了，在握手响应返回给modifyHandshake()方法之前，响应头就被适时地写入响应中。重写了modifyHandshake()方法的开发人员通常会插入某种实现或者特定于应用的HTTP头信息以进一步限定建立起的WebSocket连接。例如，或许WhipImages'R'Us WebSocket应用希望在握手时对它能接收的WebSocket消息的最大长度达成一致。该实现可以选择定义一个名为WhipImagesRUs_Max-Message-Size的HTTP头消息，表示它能处理的客户端发送的WebSocket消息的最大字节数。WhipImages'R'Us服务器端的实现将会作出响应，确认这个限制或者使用相同的HTTP头设置一个更低的限制。在这种情况下，WhipImages'R'Us应用会提供自定义的ClientEndpointConfig.Configurator和ServerEndpointConfig.Configurator，在握手期间插入合适的WhipImagesRUs_Max-

Message-Size头消息。

在我们离开 ServerEndpointConfig.Configurator 的学习时，再次注意到可以自由地使用自定义实例变量或者方法来扩展 ServerEndpointConfig.Configurator 类。

注意：

用自己的数据和方法扩展 ServerEndpointConfig.Configurator 是一个很好的方式，它使得公共信息和行为能被所有属于相同 ServerEndpointConfig 的端点实例共享，无论这些端点实例是注解式端点还是编程式端点。

4.4 WebSocket Session

如果说 EndpointConfig 代表了 WebSocket 端点的所有连接端的公共状态和算法规则，那么 Session 对象则代表了单个 WebSocket 连接的所有公共状态。客户端端点在同一时刻只允许有一个 Session 对象，这是因为客户端 WebSocket 端点只能连接一个服务器端端点。服务器端端点可能会关联多个 Session 对象，因为可能它会与多个客户端连接相关。无论哪种方式，理解使用 Session 对象的可能性是非常重要的，先从它的生命周期开始。

WebSocket Session 的生命周期

Session 对象是在成功创建 WebSocket 连接时被创建的。它有一个唯一的标识符，可通过 getID()方法获得。WebSocket 实现传递新创建的 Session 对象给打开事件处理方法(在注解式端点情况下，会用@OnOpen 注解标记该方法；在编程式端点情况下，会调用 onOpen(Session session, EndpointConfig config)方法)。此时，

Session 的访问是开放的，它是有效的。在 Chat 示例和其他章节中我们已经看到过它的一些功能，这里我们总结一下。

1. 管理 MessageHandler

Session 对象存放了用于拦截本连接上传入的 WebSocket 消息的 MessageHandler。对于注解式端点，通过方法(可能有多个)的方法签名使用@OnMessage 注解隐式地定义消息处理器。对于编程式端点，Session 的 MessageHandler 被如下的 3 个方法所管理：

```
public void addMessageHandler(MessageHandler handler)
public Set<MessageHandler> getMessageHandlers()
public void removeMessageHandler(MessageHandler handler)
```

通过这些方法分别可以添加、获取和删除 MessageHandler。正如我们所见，通常编程式端点在 onOpen()方法调用期间添加消息处理器，在 Session 中并没有作进一步的管理。然而，基于一个特殊客户端的不断变化的需要，有些 WebSocket 应用可能会在 Session 生命周期的其他点上动态地切换 MessageHandler 类。

2. 获取 RemoteEndpoint

我们已经看到，端点在需要给它的已连接端发送消息时，必须获得对方的引用，即 RemoteEndpoint(RemoteEndpoint.Basic 或 RemoteEndpoint.Async)。可以使用如下的方法获得：

```
public RemoteEndpoint.Basic getBasicRemote()
public RemoteEndpoint.Async getAsyncRemote()
```

我们已经看到过很多关于 RemoteEndpoint.Basic 对象的使用。RemoteEndpoint.Async 对象允许异步发送 WebSocket 消息，在第 5 章中会展开介绍。

3. 连接属性

Session 对象存放了一些描述底层连接的属性，其中一些属性是 Session 被创建后就不可以变更的。一个示例是使用中的子协议，除非重新连接，否则它是不能改变的，因为它是客户端和服务器在开启握手期间协商的唯一结果。另外一些属性在 Session 创建后也可以改变，如最大空闲时间属性，该属性管理底层连接允许的最大空闲时间，超时后会被 WebSocket 实现关闭。与其分步介绍所有这些属性的 Session API，不如汇总成表，参见表 4-4。

表 4-4　Session 的连接属性

属　性	用　途	是否可变
maxBinaryMessageSize	传入的二进制消息的最大长度	是
maxIdleTimeout	最大空闲时间	是
maxTextMessageSize	传入的文本消息的最大长度	是
negotiatedExtensions	在握手期间协商好的 WebSocket 扩展列表	否
negotiatedSubprotocols	在握手期间协商好的 WebSocket 子协议列表	否
protocolVersion	使用的 WebSocket 网络协议的版本	否
userPrincipal	代表已认证端用户的 Java UserPrincipal	否
secure	底层连接是否加密	否
userProperties	在 Session 中存放信息的 Map	是

4. 路径属性

本章前面已介绍过，服务器端点可以被映射成一个 URI 或者 URI 模板(级别 1)，在第 6 章将会详细地介绍 Java WebSocket API 中的路径映射机制。当向一个特定 URI 发起握手时，如果有一个匹配的 URI 模板，这个匹配会产生一个路径参数列表，可变段对

应于传入 URI 的特定值。传入 URI 也可能在请求参数中存放了查询字符串。传入 URI 的所有这些方面都可以从 Session 对象中获得，具体参考表 4-5。

表 4-5 从 Session 对象中获取请求 URI 属性的方法

方 法	用 途
URI getRequestURI()	获得完整的相对于 Web 应用上下文根的相对路径
Map<String,String> getPathParameters()	当请求 URI 匹配了一个 URI 模板时，返回名-值对的 Map
String getQueryString()	请求 URI 中的查询字符串
Map<String,List<String>> getRequestParameterMap()	从查询字符串中提取出的请求参数

这里有一些示例。如果服务器端点映射成：

```
/pets/cat
```

客户端使用如下 URI 进行连接：

```
/pets/cat?color=black&fur=long
```

则请求 URI 是/pets/cat?color=black&fur=long，查询字符串是 color=black&fur=long，请求参数是(color, black)和(fur, long)。因为端点被映射为精确的 URI，所以路径参数为空。

如果服务器端点映射成：

```
/travel/{mode}/{level}
```

客户端用来连接的传入 URI 是：

```
/travel/air/gold
```

则请求 URI 是/travel/air/gold，查询字符串和请求参数为空，路径参数是(mode, air)和(level, gold)。

5. Session 管理

管理 Session 有 3 个必需的方法。isOpen()方法可以判定一个 Session 是否还代表着一个打开的连接；getOpenSessions()方法可以列出端点所有已打开的连接；close()方法则可以关闭连接。

close()方法有两种形式：第一种是无参数的，直接关闭连接，没有特别的描述信息；第二种是允许传入一个关闭原因(用 CloseReason 实例描述)，因为连接关闭时这个信息会传递给对方，所以包含关闭原因是一个比较好的做法。

注意：

一个打开的Session并不保证底层连接是活动的，它仅仅意味着WebSocket实现并没有收到任何信息说连接已经关闭，关闭连接时有可能没有告诉任何人！如果你想知道一个Session是否代表一个活动的连接，就需要发送一个PingMessage，然后等待确认的PongMessage消息。事实上，在稳定的网络环境中，一个打开的Session通常是一个不错的选择，可以通过它来发送和接收消息。

6. 关闭连接

可能有多种原因引起连接关闭。调用 close()方法是一种优雅的关闭 Session 的方式，对端点的另一端会有响应。在这种情形下，在底层连接被关闭之前，会调用本地端点的 close()方法。Session 被关闭也可能是因为底层连接出现了问题，连接实际上已经关闭了。在这两种情况下，close()处理方法都会被调用。此时，对于使用 Session 对象要谨慎。

然而，必须注意的是，close()处理方法的实现基于 Session 是如何被关闭的，这就意味着你不能指望从 close 处理方法中发送任何最终的消息。但是，在 close()方法中，能够从 Session 对象的用户属性 Map 中获取任何已存储的应用对象，以及 Session 的其他任意的可读属性，如 Session ID 或者超时间隔。

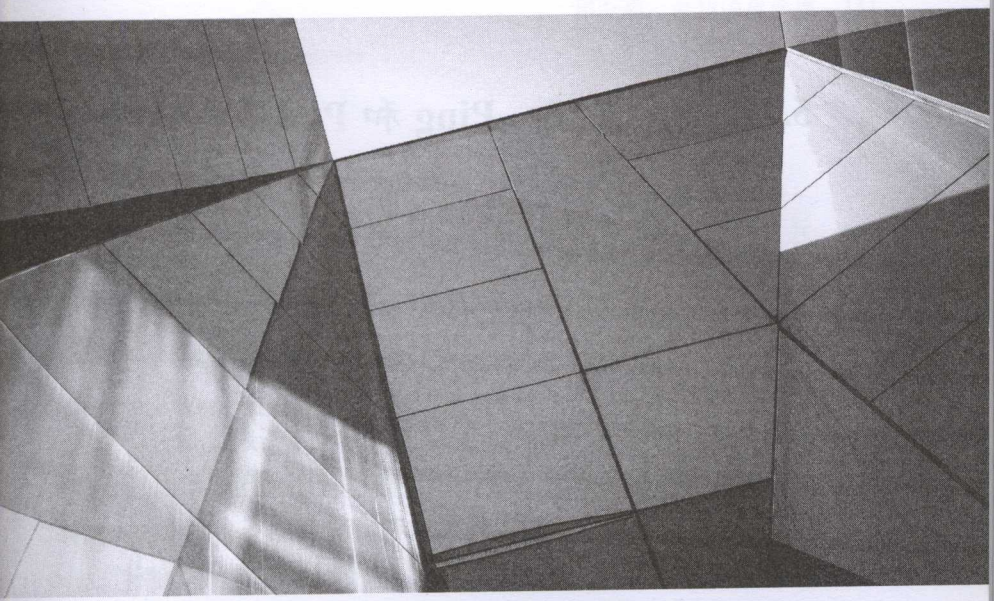
一旦端点的 close()处理方法被调用，Session 就会关闭，你将不能再访问它。事实上，如果 Session 实例已经被关闭，调用 Session 的大多数方法时都会抛出一个运行时异常。

提示：

要想在 Session 对象的方法变得不可访问之前清除掉 Session 对象中存储的任何状态，使用 close()处理方法是一个很好的主意。

4.5 本章小结

在本章中主要学习了 Java WebSocket API 的两个核心对象：Session 代表了两个 WebSocket 端点之间的一个活动连接，EndpointConfig 则被用来配置 WebSocket 端点，它存放了能被重写的信息和方法。通过 Chat 示例应用可以看到 Session 和 EndpointConfig 对象分别是如何存放应用状态的，前者存放某个连接端的应用状态，而后者存放 WebSocket 端点所有连接端的应用状态。



第5章

高级消息处理

本章将介绍 Java WebSocket API 中更专业的消息处理形式。第 3 章中讨论了比较简单的同步方式的消息发送，以及接收消息的一些选项。本章将带你了解异步形式的消息发送、使用缓冲区来接收输入消息、发送操作时的可配置超时设置以及批处理技术。我们将使用 MessageModes 应用作为示例来阐述用于消息处理的高级选项。

5.1 检查连接：Ping 和 Pong

在第 2 章中，我们接触了 Ping 消息和 Pong 消息的话题。Ping-Pong 交互只是发送一类特别的称为 Ping 消息的 WebSocket 消息，然后接收返回的另一类(尽管几乎是完全相同的)称为 Pong 消息的 WebSocket 消息。出于某些原因，你也许希望使用 Ping-Pong 交互。

首先，初始化 Ping-Pong 交互是一种判断连接是否实际工作的临时应急方式。尽管在关闭事件处理程序关闭连接时应该能够通知到你，但也有一些情况导致这种情况可能不会发生——例如，如果底层套接字由于硬件故障已经停止响应时。

第二，你也许想使用它们来保持连接存活，否则在长时间的非活跃状态下连接可能会被关闭。从第 4 章中你应该知道，如果愿意的话，可以配置 Session 对象的超时属性，但是你可能没有 WebSocket 连接的另一端的控制权——例如，如果连接的另一端是非 Java 实现的 WebSocket 协议，如浏览器。在这种情况下，发送 Ping 消息并且等待 Pong 响应将充当阻止两端超时的手动方式。

第三，你也许想通过对 Ping-Pong 消息交互占用多长时间进行计时来评估连接的健康性。这就给了你关于连接可能有多少时延的一个简单的指导。Ping 和 Pong 消息仅仅会携带非常小的数据负载，因此 Ping-Pong 交互也许并不是应用中往返流动的大消息有多迅速的一个可靠的指导，但是它能够提供一些线索。

Ping 消息和 Pong 消息都最多能够携带 125 字节的二进制数据。事实上，响应 Ping 消息的 Pong 消息必须同 Ping 消息中携带的二进制数据完全相同。

在Java WebSocket API中，发送Ping消息通过RemoteEndpoint

的如下方法进行：

```
public void sendPing(ByteBuffer applicationData)
```

若发送消息的时候发生错误，此方法将抛出 IOException 异常。你应该记得在第 2 章中，曾以两种方式中的一种侦听通过你的对等节点另一端响应 Ping 消息时返回的 Pong 消息。若编写注解式端点，则需要声明消息处理方法，此方法使用 javax.websocket.PongMessage 作为其参数，如代码清单 5-1 所示。

代码清单 5-1：Pong 消息处理方法示例

```
@OnMessage
public void catchPong(PongMessage pongMessage, Session session) {
    // ...
}
```

或者，若编写编程式端点，则应该创建实现 MessageHandler.Whole<PongMessage>接口的消息处理程序类，并且将它配置在 Session 对象上，该对象表示用于发送 Ping 消息的连接。

此时，你可能想知道，在接收 Ping 消息的接收端是否需要对 WebSocket 应用做一些特殊的事情，以便在有人发送消息时你的应用可以作出合适的响应。答案是你不需要做任何特殊的事情：所有的 Java WebSocket API 实现都要求一旦接收到入站 Ping 消息都需要立即响应合适的 Pong 消息。你无须编写任何代码来侦听入站 Ping 消息以及在响应中发送 Pong 消息。

对于 Pong 消息，最后的用途是发送其作为一类单向的心跳：它是保持连接存活(或者是嗅探失效的连接)的另一种方式。在这种场景下，可以通过 RemoteEndpoint 接口的如下方法发送 Pong 消息。

```
public void sendPong(ByteBuffer applicationData)
```

WebSocket 对等节点中连接的另一端没有义务来响应一个主动提供的 Pong 消息，所以如果发送了 Pong 消息就不要期待有回复。

如果你希望侦听主动提供的 Pong 消息，可以使用与侦听响应 Ping 消息的 Pong 消息完全一样的方式来实现，如前面所示。

5.2 异步发送 WebSocket 消息

目前为止，我们在例子中仅使用了同步模式来发送消息。也就是说，例子中所有使用的发送方法都将以某种方式阻塞，直到消息的负载传输完成。这种方式对于一些应用很有用，特别是发送小消息的应用，已经足够完美。但是，也有很多其他应用希望即使在 WebSocket 消息发送时也能够以其他方式继续工作。特别是那些发送大消息的应用更是倾向于此种方式，它可以在发送消息的同时用其处理能力继续制定其下一条消息。照片处理应用就是这种类型应用的一个极好的示例。换句话说，有一类应用在初始化线程上消息的发送时，也能够负责处理应用的其他必要功能。客户端 GUI 应用也是此种类型。它通过一个线程发送 WebSocket 消息，同时它也负责根据应用的其他输入持续更新用户界面。如果应用线程在 WebSocket 消息传输完成之前都一直阻塞，同时也冻结用户界面，这对用户来说是不可接受的事情。

现在我们将注意力放到 RemoteEndpoint.Async 接口上。此接口提供了 WebSocket 连接的远程对等节点的另一种视图。此视图允许以多种方式来发送消息，而不用等待消息被完全传输。可以通过如下方法从关联的 Session 对象获取 WebSocket 连接的一端

的 RemoteEndpoint.Async 视图的引用。

```
public RemoteEndpoint.Async getAsyncRemote()
```

RemoteEndpoint.Async 接口提供了两种方法来异步发送消息：通过 Future 或者使用 Handler。

5.2.1　通过 Future 发送 WebSocket 消息

第一种异步发送 WebSocket 消息的方法被称为“通过 Future 对象发送 WebSocket 消息”。在这种方式中，当调用发送消息的方法时，获得一个 java.util.concurrent.Future 对象的引用。例如，通过 Future 对象发送 WebSocket 文本消息的方法调用如下所示：

```
public Future<Void> sendText(String text)
```

当该方法调用返回时，它返回了一个Future对象，而此时你传递的消息很可能并没有实际发送出去。Future对象的目的是允许你追踪通过调用方法初始化的发送行为的进度。这有点类似于你在网上购买商品时的快递过程：你获取了一个追踪号码，通过此号码可以追踪包裹在路途中的进度。可以通过此Future对象的Future.isDone()方法获取消息是否已经发送。也可以通过Future.cancel()方法来取消发送行为(假设消息发送还没有完成)。还可以通过Future.get()方法来等待发送任务的完成，此方法将阻塞当前线程直至消息被完全传输。如果消息传输过程中出现了错误，只有通过Future对象才能够显式获取错误。这也正是通过Future发送WebSocket消息不会像其同步方法RemoteEndpoint.Basic那样抛出异常的原因。通过调用Future对象的get()方法可以判断消息是否被正确传输。如果传输过程中没有问题，此方法不返回任何值。如果传输过程中出现问题(可能是网络故障，也可能是消息发送时

间过长)，此方法将抛出java.util.concurrent.ExecutionException异常。实际创建的异常封装了消息传输过程中的错误，可以通过调用ExecutionException的ExecutionException.getCause()方法来进行获取。

通过 RemoteEndpoint.Async 接口，有 3 种 API 调用方式来通过 Future 发送 WebSocket 消息：一种发送文本消息、一种发送二进制消息、一种发送开发人员自定义对象。如代码清单 5-2 所示。

代码清单 5-2：通过 Future 发送消息的方法

```
public Future<Void> sendText(String text)
public Future<Void> sendBinary(ByteBuffer data)
public Future<Void> sendObject(Object data)
```

3 种API调用都返回满足前面描述的语义的Future对象。如果你希望拥有通过Future发送的自定义对象，则必须为此对象配置合适的Encoder，使得WebSocket实现能够在传输之前把自定义对象转换成本地WebSocket消息，此过程与为同步发送操作配置Encoder的方法一样。与同步方法调用一样，此处的Encoder也是在传输之前被调用。唯一不同的地方是发送API调用在编码和传输步骤之前就返回。如果编码器在给对象编码的时候抛出一个异常，此错误通过Future对象抛出。也就是说，它由ExecutionException封装，且ExecutionException能够通过Future的get()方法获得。

这里并没有特别的 API 来通过 Future 发送 Ping 和 Pong 消息，但是不用担心。因为这些消息的负载非常小，它几乎对发送没有影响：在大多数实际的设置中，在初始化发送消息和实际传输之间的时间有可能非常快。

5.2.2　通过 Handler 发送 WebSocket 消息

第二种异步发送 WebSocket 消息的方式称为“通过 Handler 发送 WebSocket 消息”。当使用这种模式发送 WebSocket 消息时，需要提供一个回调对象。例如，如果希望通过 Handler 发送一个二进制 WebSocket 消息，应该调用 RemoteEndpoint.Async 的如下方法

```
public void sendBinary(ByteBuffer data, SendHandler handler)
```

将消息数据和一个已经实现的 SendHandler 回调接口的实例传入。此方法立刻返回，并且在你提供的 WebSocket 消息传输之前。在传入消息的传输过程中，WebSocket 实现将调用 SendHandler 实现对象，使得你能够了解其传输进度。SendHandler 接口的代码如代码清单 5-3 所示。

代码清单 5-3：SendHandler 接口

```
public interface SendHandler {
   public void onResult(SendResult result);
}
```

此接口只有一个单一方法，你可以选择将其实现成一个标准的 Java 类，或是 Java SE 8 中的 Lambda 表达式。若消息传输成功，WebSocket 实现将在传输发生之后调用 SendHandler 实现中的 onResult()方法，调用方法时使用 SendResult 对象，其状态是 OK 状态。可以通过调用 SendResult 的 isOk()方法判断此状态，在此种场景下将为 true。如果在传输消息的过程中发生错误，WebSocket 实现将调用 SendHandler 实现对象的 onResult()方法，调用方法时使用 SendResult 对象，但其状态不是 OK 状态。其中的异常可以通过调用 SendResult 对象的 getException()方法获取。

对于 RemoteEndpoint.Async 接口，有 3 种通过 Handler()发送 WebSocket 消息的 API 调用方式：一种发送文本消息、一种发送二进制消息、一种发送开发人员自定义对象。如代码清单 5-4 所示。

代码清单 5-4：通过 Handler 发送消息的方法

```
public void sendText(String text, SendHandler handler)
public void sendBinary(ByteBuffer data, SendHandler handler)
public void sendObject(Object data, SendHandler handler)
```

3 个方法中的每个都在传输消息之前返回。你传入的 SendHandler将在传输过程中有任何问题时被调用。当使用 sendObject()方法时，与使用Future的sendObject()方法和同步 sendObject()方法一样，WebSocket实现将寻找合适的编码器将自定义对象转换成本地WebSocket消息。任何在实际传输消息之前的转换过程中发生的编码错误都会通过包含EncodeException异常的 SendResult对象报告给SendHandler。如果一切顺利，无论使用哪个消息类型和选择哪个方法，你传入的SendHandler都将被调用，同时其SendResult会设置为OK状态。

同通过 Future 发送一样，通过回调 Handler 来发送 Ping 和 Pong 消息也并没有什么特殊方法。

5.2.3 何时通过 Future 发送以及何时通过 Handler 发送

如果你设计的应用需要异步发送消息，那么将面临使用哪种模式的选择：通过 Future 还是通过 Handler。两种模式都提供了关于发送状态的完全一致的信息：它们通知你消息已经被传输、通知你传输消息过程中是否产生了某些类型的错误。通过 Future 发送允许你通过取消消息传输来干预传输过程，然而通过 Handler 发送并没有给你这种控制。

在某种程度上，你选择哪种模式依赖于个人编程风格以及你愿意允许你的应用线程在消息是否已经发送的监控任务上有多少消耗。如果使用通过 Future 的处理方式，你将在发送过程的某个阶段使用一个线程，它要么是检查 Future 对象来了解消息是否被传输，要么是阻塞 get()调用直到消息被发送或者是抛出指示传输失败的错误。在你初始化发送的线程中，在你希望确认返回之前，可能只有很少的事情需要完成。在这种场景中，调用通过 Future 发送的方式、注意这两件事情并且调用 Future 对象的 get()方法来阻塞线程也许是完美优雅的处理方式。或者，你用来初始化发送的线程可能负责一些进行中的任务进行长时间处理，这些任务从不等待应用中的任何一部分。在这种情况下，你不希望让线程在 Future.get()方法调用上等待。如果你使用通过 Handler 发送消息的处理方式，WebSocket 实现就负责使用其拥有的一个线程来通知你其尝试传输消息的结果的事件(成功还是失败)。例如，如果你在一个带有用户界面的应用中发送消息，你也许决定不希望将任何用户界面线程的时间奉献在等待响应中，因为用户界面线程有连续的任务来响应其他种类的异步输入，如用户调整窗口大小。在这种场景中，你应该希望发送过程结果的通知发生在另外一个线程上。因此 WebSocket 实现使用其线程之一为你提供 SendHandler 回调的方式将是非常方便的选择。

5.2.4　异步发送超时

当使用异步发送调用 RemoteEndpoint.Async 时，发送过程中一个最后层次的控制是设置超时的能力。当然，你可以自己监控发送消息所花的时间，但是它涉及用额外的编程来监控已经过去的时间量以及尝试取消发送操作。更加方便的方式是使用如下方法：

```
public void setSendTimeout(long timeoutmillis)
```

设置其参数为正值，用于表示这是你准备允许消息传输的时间上限。如果消息传输所花的时间比方法中设置的上限要长，在通过 Handler 发送方式处理时，将产生 TimeoutException 异常并传递给 SendResult；在通过 Future 方式处理时，它将被包装在通过 Future 的 get()方法抛出的 ExecutionException 中。如果你不想无限地等待通过 Future 发送或者是通过 Hanlder 发送的消息，可以设置这个超时的值为任何非正值。

5.3 消息批处理

一些更加复杂的 WebSocket 实现拥有一项称为消息批处理的特性。此特性使得 WebSocket 实现能够收集一个连接上的若干输出消息，直到收集消息的大小到达某个临界水平。在那时，即使应用认为它已经完成了消息的发送，但 WebSocket 实现事实上并没有将消息发送到连接上。当消息的采集(也被称为批消息)到达一个临界水平时，WebSocket 实现才事实上传输这些消息。当然，WebSocket 实现将按照应用打算发送的顺序发送消息。此种处理方式可以提供相当大的性能改进，特别是应用在短时间内发送大量消息时。

如果 WebSocket 实现支持批处理，则那些希望利用可能的性能提升的 WebSocket 应用必须显式告知 WebSocket 实现，它们希望允许使用此技术。它们通过调用下述方法完成此功能。

```
RemoteEndpoint.setBatchingAllowed(boolean batchingAllowed)
```

由于此选项的默认值使得在 RemoteEndpoint 上进行消息发送

时不使用批处理，如果应用永远不调用此方法来允许应用批处理技术，即使实现支持它，应用的消息都不会被批量发送。这也意味着如果你不关注消息批处理，无论你使用的 WebSocket 实现是否支持，只要从未允许应用使用此选项，它将永远不会在应用的消息上使用。

然而，一旦你允许 WebSocket 实现来批处理你的消息，就需要了解在任何的时间点上，可能在一个批处理中有一些未发送的消息还没有写入 WebSocket 连接中。因此当编写利用批处理的优点的应用时，它也给了你一个额外的需要清除的障碍。为了确保在批处理模式下没有未发送的消息，可以调用下述方法来发送任何未发送的消息：

```
RemoteEndpoint.flushBatch()
```

当允许 WebSocket 实现使用消息批处理技术时，可以看到许多性能改进很可能高度依赖于一系列的因素，包括如下：

- 应用发送的消息大小
- 应用发送消息的频率
- 应用发送消息的特定网络
- 接收消息的对等端 WebSocket 实现的性能特征

因此，为了获取最好的结果，很可能需要做大量的实验来决定是否值得采取额外的步骤，从而允许在应用中使用这项技术。

如果在不支持批处理技术的 WebSocket 实现中运行使用了 API 允许批处理的应用，此 API 调用将没有任何效果。应用能够正常运行，但是不会有消息被批处理。

如果 WebSocket 容器支持消息批处理，则它操作的层次低于 Java WebSocket API 中的异步发送 API。为了更充分地描述此处的意义，考虑一个允许批处理的应用，它运行在一个支持消息批处

理特性的 WebSocket 实现上。如果应用使用异步发送操作，在以下两种情况下可以认为其已经完成。

- 如果写入批消息不会导致此批消息写入底层连接，则一旦消息写入批消息中就认为操作完成。
- 如果写入批消息将导致整批消息都写入底层连接，则只有在整批消息(包括最后的消息)都已经写出时才认为操作完成。

这意味着 SendHandler 的回调以及 Future.get()方法的返回在写入批处理时将会发生，它可能实际上将消息写入底层连接，也可能没有写入。

5.4 缓冲、消息分片和数据帧

你应该记得在第2章中介绍过，WebSocket协议可以将WebSocket文本或二进制消息拆解成一系列的小数据帧，使得可以基于自身能力和网络能力在WebSocket连接上进行实际的发送。在WebSocket实现中使用的数据帧和通过API表示的消息间可以存在关联，也可以不存在关联。特别地，如果你选择通过类似于如代码清单5-5的方式声明一个方法以分片接收消息，则实际上通过此方法接收的片段与WebSocket实现通过WebSocket连接发送的片段可能并不完全一致。

代码清单 5-5：以分片形式处理消息的方法示例

```
@OnMessage
public void handleMessagePieces(String piece, boolean isLast) {
    // process each piece until the last piece arrives
}
```

例如，WebSocket 实现可能选择多个入站数据帧缓冲到一个分片中。同样，如果选择以一个分片接收所有 WebSocket 消息，则示例如下所示：

```
@OnMessage
public void handleMessage(String wholeMessage) {
    //
}
```

连接的对等端会给你发送一个 WebSocket 文本消息，它以一系列的多个数据帧的形式抵达，每个数据帧都包含完整消息的某个较小的分片。WebSocket 实现没有其他选择，只能缓冲这些入站消息分片，直到最后一个消息分片抵达，然后它将此拼图的所有分片装配到一起并通过上述方法分发完整的消息。

这样，WebSocket 实现支持以某种形式来缓冲入站消息。对于处理大消息的应用、期望有多个客户端发送消息的应用或者是遇到行为恶劣的客户端偶尔尝试发送不合常理的巨大消息的应用(或者是 3 种情况都一起发生)，此类缓冲被证明也许是个问题。它可能证明在计算资源上花费非常昂贵，或者你的应用可能并不是为处理如此大的消息而设计的。因此，你可能很乐意让 WebSocket 实现为你控制缓冲级别。

在 Java WebSocket API 中，有两个控制手段，通过它们可以在上述描述的情况下设置缓冲级别的限制。第一个编程式的处理方法允许直接控制每个单独连接的缓冲区大小限制，不管连接上是文本消息还是二进制消息，都可以分别使用 Session 对象上的下述方法进行控制。如代码清单 5-6 所示。

代码清单 5-6：控制入站消息缓冲区的会话方法

```
public void setMaxBinaryMessageBufferSize(int length)
public void setMaxTextMessageBufferSize(int length)
```

一旦设置了缓冲区大小，导致缓冲区溢出的任何入站消息将不会被传递。例如，当应用选择以完整形式接收消息，一个超过缓冲区的容量的长序列的入站消息分片用于重建消息时，其将不会被传递。在此场景下，将会调用WebSocket端点的错误处理方法，连接将被关闭，同时会用专用的CloseReason.CloseCode.TOO_BIG代码来描述其关闭的原因。

另一个可以用来处理注解式端点的机制是，可以设置传递给带有注解@OnMessage 的方法的消息大小的限制。考虑如下示例代码清单 5-7。

代码清单 5-7：带有容量限制的消息处理程序示例

```
@OnMessage(
  maxMessageSize= 64 * 1024
)
public void handleSmallMessages(byte[] data, Session session) {
    // messages are always 64kb or under.
}
```

此处限制了可以传递给方法的消息的大小为 64KB。

注意：

设置 Session 对象的缓冲区大小可以让你控制每个底层连接使用的缓冲区大小。可以为应用中的每个连接设置不同的缓冲区大小。消息大小的限制应用于由 WebSocket 端点通过任何客户端处理的消息。

5.5 保证消息传递

WebSocket 协议并不能保证所发送的 WebSocket 消息事实上

能够被连接的另一端正确地接收。在大多数情况下，如果传输消息过程中发现错误，WebSocket 实现将会知道，但是消息并没有传递到另一端且没有任何错误给出的情况也有可能发生。因此，当考虑 Java WebSocket API 的发送操作时，知道发送操作的结果是告知你消息被传输但是不一定被接收是非常重要的。如果你需要确切地知道应用中的 WebSocket 消息被接收者所接收，就应该考虑在编程时处理某种回执，使得接收者发送事实上它收到了发送者发送的消息的确认。幸运的是，通过@OnMessage 注解创建消息处理方法，返回一个简短的回执是件相当容易的事情：仅仅在方法中包含一个返回值，WebSocket 实现会立刻返回它。

5.6　发送消息 API 总结

最后，我们使用一个表格来作为这部分的总结。表 5-1 概述了使用 RemoteEndpoint 及其子类发送文本、二进制以及 Ping 和 Pong 消息的所有选项。

表 5-1　发送消息 API

发送类型和负载	模　式	接　口	API
Ping 和 Pong	Sync	RemoteEndpoint	void sendPing(ByteBuffer b)
文本	Sync	RemoteEndpoint.Basic	void sendText(String text)
文本	Sync、Sequence	RemoteEndpoint.Basic	void sendText(String textPart, boolean isLast)

(续表)

发送类型和负载	模　式	接　口	API
文本	Async、Future	RemoteEndpoint.Async	Future<Void> sendText(String text)
文本	Async、Handler	RemoteEndpoint.Async	void sendText(String text, SendHandler handler)
二进制	Sync	RemoteEndpoint.Basic	void sendBinary (ByteBuffer b)
二进制	Sync	RemoteEndpoint.Basic	void sendBinary(ByteBuffer dataPast, boolean isLast)
二进制	Async、Future	RemoteEndpoint.Async	Future<Void> sendBinary(ByteBuffer b)
二进制	Async、Handler	RemoteEndpoint.Async	void sendBinary(ByteBuffer b, SendHandler handler)
Java 对象	Sync	RemoteEndpoint.Basic	void sendObject(Object data)
Java 对象	Async、Future	RemoteEndpoint.Async	Future<Void> sendObject(Object data)
Java 对象	Async、Handler	RemoteEndpoint.Async	void sendObject(Object data, SendHandler handler)

5.7 MessageModes 应用

现在，我们已经查看了 Java WebSocket API 中很多有关消息

处理的更加高级的话题，是时候立刻开始介绍一些代码了。MessageModes 应用是一个客户端/服务器类型的 WebSocket 应用，通过该应用可以练习到目前为止本章已介绍的许多高级消息处理特性。

5.7.1　MessageModes 应用概述

应用的服务器端是一个简单的、注解式的服务器端点，其工作是处理入站文本和二进制消息，然后将此接收到的消息报告返回给客户端。

应用的客户端是一个 Java Swing 应用，它使用注解式客户端端点来测试 Java WebSocket API 的所有发送模式。可以发送可配置大小的文本和二进制消息。应用中可以用来发送文本和二进制消息的模式如下：

- 作为完整消息同步发送
- 作为消息分片序列同步发送
- 通过 Future 异步发送
- 使用 Handler 异步发送

可以配置异步发送操作的超时属性，并且能够看到服务器端点收到消息后作出的响应。最后，可以通过初始化一个定时的 Ping-Pong 交互来测试连接的健康性。

图 5-1 展示了客户端应用的外观。

当选择屏幕最顶端的消息时，可以调整 Size 滑动条来制定文本消息或者是二进制消息，具体取决于你的选择，其范围约在 300KB~15MB 之间。你应该注意到，当一直移动滑动条直到最右端时，用户界面将通知你，将发送的消息超过了服务器端点的最大大小限制。

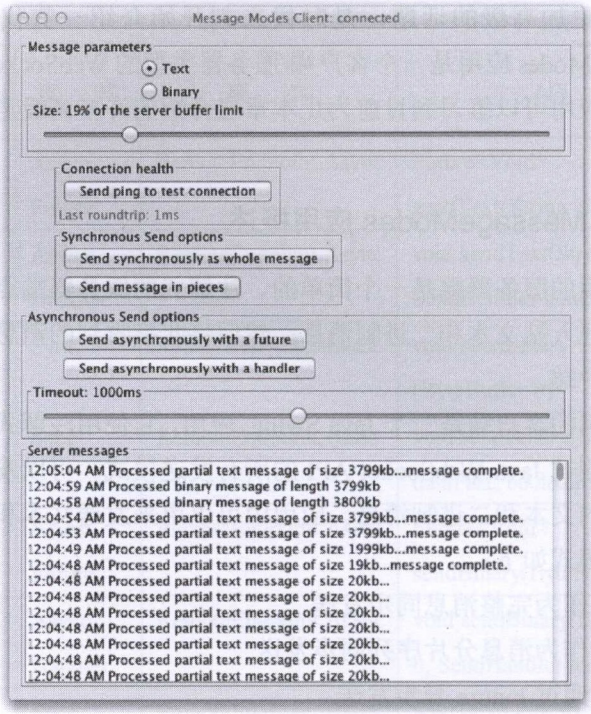


图 5-1 Message Modes 窗口

当按下 send ping 按钮时，客户端应用制定一个 Ping 消息并将其发送。当消息返回时，客户端应用记录发送消息所花费的时间并更新用户界面。

此处有两个按钮用于同步地发送消息，一个按钮用于发送完整消息，另一个按钮将消息作为由 100 个小分片构成的序列进行发送。

此处还有两个按钮用于异步地发送消息，一个通过 Future 发送消息，另一个通过 Handler 发送消息。在这两个按钮下面，可以看到一个滑动条，通过此滑动条可以配置异步发送超时参数，

其范围在 0~2 秒之间，其中 0 秒表示发送操作永远不会超时。

最后，窗口底部的文本区显示发送的任意服务器更新。

当运行此应用时，可以花一些时间来尝试所有不同的选项，从而获得 Java Web Socket API 中所有不同发送模式的大致感觉。务必尝试不同的消息大小和异步发送超时参数，看看所发生的现象。

5.7.2　查看 MessageModes 应用的代码

下面继续介绍此应用，查看使其工作的一些代码。首先，让我们看看代码清单 5-8。

代码清单 5-8：MessageModesServer 服务器端点

```
import java.io.IOException;
import java.text.DateFormat;
import java.util.Date;
import javax.websocket.*;
import javax.websocket.server.*;

@ServerEndpoint(value="/modes")
public class MessageModesServer {
  public static final int MESSAGE_MAX = 15 * 1000 * 1024;
  // 15, 000 kb

  @OnOpen
  public void open(Session session) {
    session.setMaxBinaryMessageBufferSize(MESSAGE_MAX);
    this.reportMessage(session, "Connected !");
  }

  @OnMessage
  public void binaryMessage(byte[] bytes, Session session) {
    this.reportMessage(session, "Processed binary message of
    length " + bytes.length / 1024 + "kb");
```

```
    }

    @OnMessage(
      maxMessageSize = MESSAGE_MAX
    )
    public void textMessage(String partialMessage,
            boolean isLast,
            Session session) {
      String report = "Processed partial text message of size "
          + partialMessage.length() / 1024 + "kb...";
      if (isLast) {
        report = report + "message complete.";
      }
      this.reportMessage(session, report);
    }

    public void reportMessage(Session s, String message) {
      try {
        String timeStamp =
          DateFormat.getTimeInstance().format(new Date());
        s.getBasicRemote().sendText(timeStamp + " " + message);
      } catch (IOException ioe) {
        System.out.println(ioe.getMessage());
      }
    }

    @OnError
    public void error(Throwable t) {
      System.out.println("MessageModesServer error: "
        + t.getMessage());
    }

    @OnClose
    public void close(Session s, CloseReason cr) {
      System.out.println("MessageModesServer closing because: "
        + cr.getReasonPhrase());
    }
}
```

当新的连接打开时，MessageModesServer端点做的第一件事情是设置WebSocket实现将用来缓冲入站二进制消息的缓冲器的大小限制。然后，它将一个简单消息报告给客户端，这可以被此应用的用户看到。你应该看到它有两个方法处理入站消息：一个是用于处理入站的二进制消息的方法binaryMessage()，它要求WebSocket实现以字节数组的完整形式传递此消息。第二个处理方法是textMessage()，你应该立刻注意到它使用@OnMessage注解的maxMessageSize属性，以此注解方式来限制可以传入的消息的大小。此方法选择以消息分片形式来接收文本消息。这两个方法都将给发送者报告一个通知性的消息，从而能够以用户可读的形式来确认入站消息的接收。

注意，服务器端点使用两种不同的方式来限制入站消息的大小。对于二进制消息来说，此限制设置在当容器需要缓冲入站消息时所使用的二进制数据缓冲区上。对于文本消息来说，此限制设置在可以传递给消息处理方法的消息的大小上。接下来你将看到，这些不同的方式在运行时产生的结果有一些微妙的区别。

现在，让我们看看代码清单 5-9 所示的客户端端点 MessageModesClient 类。此端点是一个注解式端点，管理 Java Swing 窗口(MessageModesWindow)和服务器端点间的所有 WebSocket 交互。

代码清单 5-9：MessageModesClient 客户端端点

```
import java.io.IOException;
import java.nio.ByteBuffer;
import java.util.concurrent.Future;
import javax.websocket.*;

@ClientEndpoint
public class MessageModesClient {
```

```
private Session session;
private MessageModesClientListener listener;
static int PIECES_COUNT = 100;
private int sendTimeout = 10;

public MessageModesClient(MessageModesClientListener
listener) {
  this.listener = listener;
}

public void setTimeout(int millis) {
  this.sendTimeout = millis;
}

@OnOpen
public void open(Session session) {
  this.session = session;
  this.listener.setConnected(true, null);
}

@OnMessage
public void handleMessage(String message) {
  this.listener.reportMessage(message);
}

@OnMessage
public void handlePong(PongMessage pm) {
  String sendAtString =
    new String(pm.getApplicationData().array());
  long sendAtMillis = Long.parseLong(sendAtString);
  long roundtripMillis = System.currentTimeMillis() -
     sendAtMillis;
  this.listener.reportConnectionHealth(roundtripMillis);
}

@OnClose
public void close(Session session, CloseReason cr) {
  this.listener.setConnected(false, cr);
```

```
}

public void disconnect() throws IOException {
  this.session.close(
    new CloseReason(CloseReason.CloseCodes.NORMAL_CLOSURE,
      "User closed application"));
}

public void sendPing() throws IOException {
  long now = System.currentTimeMillis();
  byte[] data = ("" + now).getBytes();
  session.getBasicRemote().sendPing(ByteBuffer.wrap(data));
}

public Future<Void> sendAsyncByFuture(byte[] data)
    throws IOException {
  RemoteEndpoint.Async rea = session.getAsyncRemote();
  rea.setSendTimeout(this.sendTimeout);
  ByteBuffer bb = ByteBuffer.wrap(data);
  return rea.sendBinary(bb);
}

public Future<Void> sendAsyncByFuture(String textData)
    throws IOException {
  RemoteEndpoint.Async rea = session.getAsyncRemote();
  rea.setSendTimeout(this.sendTimeout);
  return rea.sendText(textData);
}

public void sendAsyncWithHandler(byte[] data, SendHandler sh) {
  RemoteEndpoint.Async rea = session.getAsyncRemote();
  ByteBuffer bb = ByteBuffer.wrap(data);
  rea.setSendTimeout(this.sendTimeout);
  rea.sendBinary(bb, sh);
}

public void sendAsyncWithHandler(String textData,
SendHandler sh) {
```

```
        RemoteEndpoint.Async rea = session.getAsyncRemote();
        rea.setSendTimeout(this.sendTimeout);
        rea.sendText(textData, sh);
    }

    public void sendSynchronously(byte[] data) throws IOException {
        RemoteEndpoint.Basic reb = session.getBasicRemote();
        ByteBuffer bb = ByteBuffer.wrap(data);
        reb.sendBinary(bb);
    }

    public void sendSynchronously(String textData) throws
    IOException {
        RemoteEndpoint.Basic reb = session.getBasicRemote();
        reb.sendText(textData);
    }

    public void sendInPieces(byte[] data,
    PartialMessageSendListener pc)
            throws IOException {
        RemoteEndpoint.Basic reb = session.getBasicRemote();
        int chunkSize = (int) (data.length / PIECES_COUNT);
        BinaryDataIterator di = new BinaryDataIterator(data,
        chunkSize);
        int counter = 0;
        while (di.hasNext()) {
            ByteBuffer nextPiece = di.next();
            boolean isLast = !di.hasNext();
            reb.sendBinary(nextPiece, isLast);
            counter++;
            pc.reportProgress(counter);
        }
    }

    public void sendInPieces(String textData,
            PartialMessageSendListener pc)
                throws IOException {
        RemoteEndpoint.Basic reb = session.getBasicRemote();
```

```
        byte[] bytes = textData.getBytes();
        int chunkSize = (int) (bytes.length / PIECES_COUNT);
        BinaryDataIterator di = new BinaryDataIterator(bytes,
        chunkSize);

        int counter = 0;
        while (di.hasNext()) {
          ByteBuffer nextPiece = di.next();
          boolean isLast = !di.hasNext();
          reb.sendText(new String(nextPiece.array()), isLast);
          counter++;
          pc.reportProgress(counter);
        }
      }
    }
```

我们将不会列出此应用的用户界面代码，但是会介绍MessageModesClientListener接口。此接口包含了MessageModesClient端点需要提供给用户界面的更新。MessageModesWindow类拥有应用的客户端部分的所有GUI代码，它实现了此接口，从而允许MessageModesClient端点保持最新的GUI。下面介绍那些更新中的每一个方法，当然，也就是接口MessageModesClientListener中的每个方法。

```
public void setConnected(boolean isConnected, CloseReason cr)
```

这个方法在客户端到服务器的连接打开或者关闭时被客户端端点调用。当连接关闭时，它通过 CloseReason 类的实例的形式提供关闭的原因。

```
public void reportMessage(String message);
```

这个方法在客户端接收到服务器端点的消息时被客户端端点调用。GUI 使用此调用在窗口底部的文本区中填写消息。

```
public void reportConnectionHealth(long millis);
```

这个方法在接收到携带着往返花费时间的响应 Ping 的 Pong 消息时被客户端端点调用。然后，GUI 使用此消息来更新其显示。

MessageModesClient 拥有其连接的 Session 对象的本地状态。注意，因为 MessageModesClient 需要在生命周期方法之外的上下文中拥有对 Session 对象的引用(我们已经知道可以选择将 Session 对象给它们)，所以这是必要的。这与 MessageModesServer 端点形成对照。MessageModesServer 端点仅在响应入站消息时将更新发送给客户端。所以，在服务器端点中没有必要将 Session 对象存储为一个实例变量。另一个状态是超时参数，它通过用户界面的滑动条使用 setTimeOut()方法调用来控制。

MessageModesClient 的生命周期方法相当简单。打开处理方法设置会话实例变量并通知 GUI 端点已经连接。关闭处理方法通知 GUI 端点已经断开及其原因。文本消息处理方法通知 GUI 新的消息已经从服务器端到达。下面花些时间来介绍 Pong 的处理方法。

你应该注意到，MessageModesClient 类的 sendPing()方法中发送的 Ping 消息的内部应用数据是表示当前时间的字节数组。你应该记得所有 Java WebSocket 实现都有责任回复入站的 Ping 消息一个 Pong 消息，并且携带着与已经发送的 Ping 消息完全一样的负载。因此，你将会看到 Pong 处理方法的实现依赖于此要求，因为其从 Pong 消息中提取数据，并且在 Ping 消息发送后拦截它来计算往返花费时间以便通知 GUI。

MessageModesClient 类的剩余方法还包括 GUI 调用用于发送消息的 8 个发送方法。操作包括支持文本和二进制消息发送的 4 种模式：作为完整消息同步发送、作为消息分片序列同步发送、

通过 Future 异步发送和使用 Handler 异步发送。

5.7.3　MessageModes 应用中需要注意的事情

你应注意到 MessageModesClient 的两个 sendInPieces()方法，它们以一个消息分片序列的方式同步发送消息，并接受一个回调对象，即 PartialMessageSendListener 实例，用于报告发送消息分片的进度。当使用此模式发送消息时，这就是你从 GUI 中看到的进度对话框前进的动力，如图 5-2 所示。它同时阐明了使用这种发送模式的一个优点：可以在此种方式的每一步中报告进度。你也应注意到在 sendInPieces()方法中，同其他模式的另一个不同之处是 MessageModesClient 负责决定顺序发送的粒度。

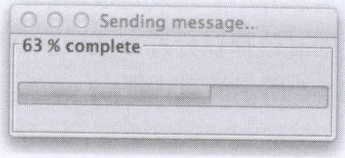


图 5-2　以分片的方式发送消息

通过 Future 发送消息的方法 sendAsyncByFuture()将 Future 对象返回给 GUI 代码，使得显示如图 5-3 所示的对话框。此对话框在你选中此模式发送消息时弹出，并允许通过 GUI 取消发送操作。

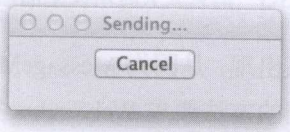


图 5-3　通过 Future 发送消息

如果你要求 GUI 通过回调 Handler 发送消息，假如消息传输成功，那么应该会看到如图 5-4 所示的对话框。

图 5-4 通过 Handler 发送消息——传输已确认

通过查看 MessageModesClient 的 sendAsyncWithHandler()方法，你应该看到 GUI 代码制定了 SendHandler 实现，从而导致消息对话框在传输成功时出现。

到目前为止一切顺利，但是如果你进一步尝试 MessageModes 应用，应该注意到消息并不总是发送成功。例如，如果你增大消息大小到最大值，同时选择以分片方式同步发送二进制消息，那么应该会看到如图 5-5 所示的对话框。

图 5-5 消息太大时所发生的情况

此对话框之所以出现，是因为 MessageModesServer 用来处理入站二进制消息序列的方法要求 WebSocket 实现以完整的字节数组传输消息。在这种情况下，运行在服务器端的 WebSocket 实现没有任何选择，只有缓冲入站二进制消息分片。然而，这些消息分片的总大小超出了 MessageModesServer 在会话中设置的最大缓冲区大小。因为入站消息太大，所以服务器端的 WebSocket 实

现会关闭连接，并且调用服务器端点的错误处理方法来通知此方法中发生的错误。MessageModesClient 将接收到关闭通知，接着通知用户界面使得弹出对话框，给用户重新连接的选择(如果用户想要连接的话)。

如果你尝试发送的文本消息超过服务器端点设置的限制，此对话框也会出现。

最后，如果你调整异步发送的超时时间为足够小的值，然后尝试异步发送消息，将看到如图 5-6 所示的对话框。

图 5-6　发送消息超时的情况

例如，当使用 Handler 和一个较低的超时时间异步发送消息时，此情况将会发生。在这种情况下，异步发送操作产生一个发送结果，此结果在操作超时时由 WebSocket 实现创建。你应该看到此情况由 MessageModesWindow.sendCompleted()方法中的如下代码清单 5-10(在操作超时时，客户端接收 sendResult 变量作为发送结果)来实现。

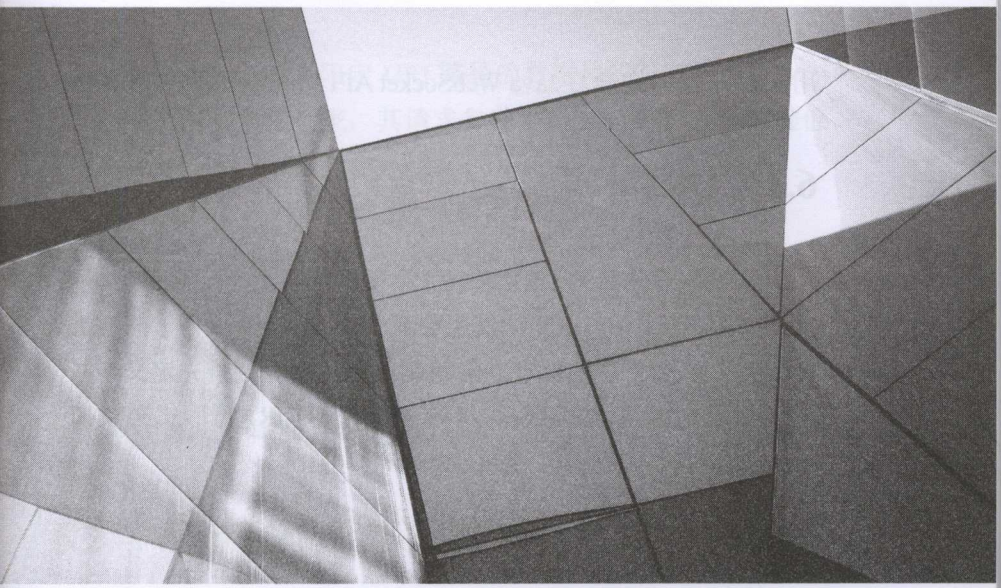
代码清单 5-10：MessageModesWindow 中处理 SendResult 的代码片段

```
if (sendResult.isOK()) {
  long millis = System.currentTimeMillis() - now;
  JOptionPane.showMessageDialog(this,
    "Message transmitted in " + millis + "ms" ,
    "Message Send Update",
```

```
            JOptionPane.INFORMATION_MESSAGE);
        } else {
            String message = "Error sending message:\n" +
                sendResult.getException().getMessage();
            Throwable ex = sendResult.getException();
            if (ex instanceof ExecutionException) {
                Throwable eex = ((ExecutionException) ex).getCause();
                if (eex instanceof
                    java.util.concurrent.TimeoutException) {
                    message = "Send timed out !\n
                        Try increasing the timeout...";
                }
            }
            JOptionPane.showMessageDialog(this,
                    message,
                    "Warning",
                    JOptionPane.ERROR_MESSAGE);
        }
```

5.8 本章小结

至此，我们已介绍完高级消息处理的主题。在本章中，我们已经学会如何使用 Java WebSocket API 来异步发送消息，可以使用 Future 对象来追踪消息传输的进度，或者通过提供传输完成时能够回调的 SendHandler 对象来实现。同时学会了如何为这些发送操作配置超时时间，以及如何配置应用来限制入站消息的大小。还学到了有些 WebSocket 实现支持消息批处理特性，以及当这些特性可用时如何利用它们。



第6章

WebSocket 路径映射

正如我们很多人可以通过各种地址(例如电子邮件、电话号码、即时通信、街道地址)取得联系一样，服务器 WebSocket 端点在托管它的服务器上的 URI 空间中也有一个地址。前面已经介绍过客户端 WebSocket 端点如何连接到服务器端点，也介绍了如何把服务器端点映射到Web服务器的名称空间上的一个简单的相对URI。本章中将介绍把服务器端点映射到托管它们的 Web 容器上的 URI 空间的所有技术，以及如何在 WebSocket 应用中更好地利

用它们。我们将学习 Java WebSocket API 路径映射的 9 条规则。

6.1 URI 术语

本书假设你熟悉 URI(统一资源标识符)。然而，依赖于环境，术语 URI、URL 和 URN 的使用会有些不正式，所以为了探索 Java WebSocket API 中的路径映射机制的能力，我们首先定义一些你应该知道的术语。WebSocket 端点的绝对 URI 的形式必须如下所示：

```
<websocket-protocol-scheme>://<authority>:<port-number>/<uri
-path>?<query-string>
```

其中：

- websocket-protocol-scheme 是 ws 或者 wss，其值取决于连接到的端点是未加密的还是已加密的。我们将在第 7 章回到安全的话题。
- authority 通常只是托管 WebSocket 端点的 Web 服务器的主机名称。
- port-number 是托管 WebSocket 端点的 Web 容器用于侦听入站打开阶段握手请求的端口号。对于 Java WebSocket API 的实现来说，它通常与 Web 服务器侦听任何类型的 HTTP 请求的端口号是一样的，一般是 8080。
- uri-path 是通过分隔符“/”分隔的由路径片段组合的字符串。路径引用某种资源——在我们的示例中，它可能是 Web 容器的 URI 空间中的 WebSocket 端点。

- query-string 是限定 URI 路径的数据片段。它的形式为一个请求参数列表，其请求参数是通过&或者;分隔的键值对：key=value。

下面是 WebSocket URI 的一些示例：

- ws://example.org/Play/tictactoe
- wss://example.org:8000/FINANCE/secure/account-updates
- ws://example.org:8080/websockets/Chat?mode=lazy&timeout=60

请求参数有时也被称为查询参数。

6.2　WebSocket 路径映射

Java WebSocket API 中有两种将服务器端点映射到路径的机制：精确 URI 映射和 URI 模板映射。我们将从你已经见过的机制开始介绍。

6.2.1　精确 URI 映射

目前为止，我们已经在本书的所有示例中见过，可以将一个服务器端的 WebSocket 端点映射到一个 URI 路径。URI 路径是上下文中的一个相对 URI，它的形式为我们前面定义过的 uri-path 的形式，并且总是以/开始。一旦确定了路径，对于注解式端点来说，就可以在 WebSocket 端点中声明它，如代码清单 6-1 所示，其 URI 路径是/tools/chat。

代码清单 6-1：注解式端点中的精确 URI 映射

```
@ServerEndpoint("/tools/chat")
public class MyChatServer {
```

```
    ...
}
```

或者，在编程式端点的情况下，当创建ServerApplicationConfig实现时，需要为端点提供一个ServerEndpointConfig的实例，如第1章所见。当使用ServerEndpointConfig.Builder创建ServerEndpointConfig时，需要提供该URI路径，如代码清单6-2所示，此处我们将此编程式端点映射到同样的路径/tools/chat。

代码清单 6-2：配置编程式端点的精确 URI 映射

```
ServerEndpointConfig sec = ServerEndpointConfig.Builder
        .create(MyChatEndpoint.class, "/tools/chat")
        .build();
```

当部署任一端点时，将端点类(在编程式端点的情况下，也包括额外的 ServerApplicationConfig)打包成 WAR 文件。当创建 WAR 文件时，需要提供一个称为 Web 应用上下文根的 URI 路径。当部署在 Web 服务器上时，此 Web 应用的上下文根定义了 WAR 文件中所有资源的 URI 名称空间的根。这样，假如 Web 服务器将 Web 应用发布到一个路径/<web-apps>下的名称空间，如下所示：

```
<hostname> /<web-apps>
```

则 WAR 文件的 URI 名称空间的根可以通过 URI

```
<hostname>/<web-apps>/<context-root>
```

来访问，其中<context-root>是 WAR 文件的上下文根。大多数 Web 服务器简单地将 Web 应用发布到它们的 URI 名称空间的根下(对于<web-apps>，它们使用空路径)。

下面是 Java WebSocket API 中 URI 映射的第一个规则。

规则 1：服务器 WebSocket 端点的 URI 路径被 Web 服务器作为相对于部署的 Web 应用的 Web 应用上下文根对待。

因此，用来访问映射到 URI 路径的服务器端点的绝对 WebSocket URI 看起来如下所示：

```
<ws or wss>://<hostname>:<port>/<webserver-root>/
   <context-path>/ <websocket-uri-path>
```

其尾部可能带有查询字符串。

因此，假如你将任一端点打包成 WAR 文件，其上下文根等于/employee，并且将其部署到 Web 服务器上以便发布 Web 应用到其名称空间的根，那么将可以通过连接

```
ws://example.org/employee/tools/chat
```

来访问端点。

你应该已经非常熟悉规则1：目前为止，本书中的所有服务器端点都使用精确URI映射。可以在各自的客户端通过绝对URL来访问它们。此URL通过将规则1应用到服务器端点的URI路径而推断出。

现在，如果你尝试将前面的示例代码片段中的注解式端点示例和编程式端点示例作为同一个 Web 应用的一部分同时部署，那么会发现 WebSocket 实现将因为规则 2 而拒绝部署此 WAR 文件。

规则 2：两个拥有相同 URI 路径的端点不能部署在同一个 Web 应用中。

我们希望此规则的理由对你来说已经很明显了：如果允许同时部署注解式和编程式端点，那么它们将被 Web 服务器发布到相同的绝对 URI 下：

```
ws://example.org/employee/tools/chat
```

因此当客户端试图连接到此 URI 时，Web 服务器无法指示哪个端点用于打开阶段握手。如果尝试部署违反此规则的应用，则 WebSocket 实现将部署失败。

当然，你可以在不改变每个 WebSocket 映射的 URI 路径的情况下，将两个端点部署到一个 Web 服务器，只需要将每个端点放到不同的 Web 应用中即可。若如此，你需要提供两个不同的上下文根，Web 服务器不会再面临尝试把两个 Web 资源发布到其名称空间中的同一个 URI 的问题。

让我们来考虑一下规则 3。

规则 3：Web 服务器处理的 URI 路径是大小写敏感的。

这意味着在同一个 Web 应用中，通过上下文根/garden 可以部署如下两个端点：

```
@ServerEndpoint("/orchard")
public class MyAppleTrees {
    ...
}
```

和

```
@ServerEndpoint("/Orchard")
public class MyPearTrees {
    ...
}
```

请求 ws://example.org/garden/Orchard 将连接到梨树，而请求 ws://example.org/garden/orchard 将连接到苹果树。

6.2.2 URI 模板路径

可以用来发布服务器端点的第二种路径称为 URI 模板。URI

模板被定义为一个紧凑的字符序列，它通过变量扩展来描述一系列的 URI。换句话说，URI 模板与 URI 类似，除了其 URI 的一部分已经被变量替换之外。在更深入地介绍 URI 模板之前，以下是相对 URI 模板的一个简单示例：

```
/travel/{access-level}/flights
```

在此 URI 模板中，其中的一个路径片段由变量 access-level 表示。URI 模板十分有用，因为将 URL 的片段表示为变量，所以一个 URI 模板实际上可以表示一系列的 URI。若 URI 模板为每个变量部分假定固定值，则其中的每一个都同 URI 模板一样。例如，URI 模板等同于如下 URI：

```
/travel/vip/flights, 当变量 access-level 是 vip 时
/travel/coach/flights, 当变量 access-level 是 coach 时
```

如果一个 URI 等价于带有特定值变量的 URI 模板，则其被描述成 URI 模板的有效扩展。

下面是一个映射成示例 URI 模板的 WebSocket 服务器端点的示例：

```
@ServerEndpoint("/travel/{access-level}/flights")
public class BookingService {
    ...
}
```

你可以很容易地猜想到通过同样的路径 /travel/{access-level}/flights 来创建 ServerEndpointConfig 的代码，它可以用于配置编程式端点。

现在，我们介绍规则 4。

规则 4：若打开阶段握手的请求 URI 是 URI 模板的有效扩展，则入站的打开阶段握手将匹配映射到此URI模板路径的服务器端点。

利用规则 4 可以看到，如果 WAR 文件中的 BookingService 端点部署在 fun.org，其上下文根是/customer/services，则客户端可以通过如下的任一 URI 连接到 BookingService：

```
ws://fun.org/customer/services/travel/vip/flights
ws://fun.org/customer/services/travel/premier/flights
ws://fun.org/customer/services/travel/executive/flights
ws://fun.org/customer/services/travel/cattleclass/flights
```

其中，变量 access-level 的值分别是 vip、premier、executive 和 cattleclass。

一般情况下，URI 模板的复杂性分成几个不同级别。你刚刚见过的是最简单的变量替换示例。还有一些更复杂的级别，其 URI 的一部分包括正则表达式，其扩展通过正则表达式匹配来进行计算。Java WebSocket API 仅允许 URI 模板的简单形式，由此引出规则 5。

规则 5：对于服务器端点映射来说，Java WebSocket API 仅允许级别为 1 的 URI 模板。

级别为 1 的 URI 模板仅允许路径变量是简单的命名变量，由{}包围，不包含任何 URI 保留字(例如+、*或者/)。这意味着如下 URI 模板的级别为 1，因此它们都能作为 WebSocket 端点的路径：

```
/{foo}
/street/{number}/{directionality}
/{color}/trees/{seasonality}/{name}
```

但是，如下表达式不是级别为 1 的 URI 模板。

```
/{*}/foo
```

```
/{+day}/pants
/map/{x,y}
/{keys*}
/location/{country/county}
```

这里以牺牲最终的灵活性为代价，为 Java WebSocket API 的匹配模式带来了一些简化。不过，由于 Java WebSocket API 仅允许级别为 1 的 URI 模板，因此从现在开始我们将使用术语 URI 模板来代替级别为 1 的 URI 模板。URI 模板的匹配规则由规则 6 所涵盖。

规则 6：当且仅当 URI 模板中的每个变量都可以使用非空的值进行扩展，并使得扩展的结果与此 URI 精确一致时，入站 URI 才匹配级别为 1 的 URI 模板。

6.2.3　URI 模板匹配相关的 API

使用 URI 模板来映射服务器端点的好处不仅限于允许通过多个 URI 连接到一个发布的端点，而且能够利用匹配的副产品 URI 模板变量的值。例如，继续前面的 BookingService 示例，如果我们能通过如下 URI

```
ws://fun.org/customer/services/travel/vip/flights
```

连接到 BookingService 端点，则可能期望比通过如下 URI

```
ws://fun.org/customer/services/travel/cattleclass/flights
```

连接到 BookingService 端点有更好的服务级别、选项(和价格)。

为了给服务级别分级，BookingService 端点需要能够访问入站 URI 匹配过程中产生的变量值。

JavaWebSocket API提供两种方式来访问此变量值。首先，对

于注解式端点和编程式端点都可用的方式是Session对象拥有这些变量，通过调用其方法

```
public Map<String, String> getPathParameters()
```

返回一个映射，其中的键是路径参数变量名称(即 URI 模板中用{}括起的字符串)，其映射值是匹配入站 URI 时获取的变量值。如代码清单 6-3 所示。

代码清单 6-3：使用 URI 模板的注解式端点

```
@ServerEndpoint("/travel/{access-level}/flights")
public class BookingService {

  @OnOpen
  public void processNewClient(Session session) {
    Map<String, String> pathParameters =
      session.getPathParameters();
    String memberLevel = pathParameters.get(
        "access-level");
    switch (memberLevel) {
      case "vip":
        this.offerBestSeatsAndPriorityDeals();
      default:
        this.offerCheapSeats();
    }
  }
...
}
```

如果开发注解式端点，则获取入站URI中的路径参数的第二个选择是通过使用@PathParam注解。@PathParam有一个单一的值属性，它是需要查找的值的路径参数的名称。对于任意生命周期方法(通过@OnOpen、@OnMessage、@OnError和@OnClose注解的方法)，通过包含用携带URI模板中的变量名称的@PathParam 注

解标记的String类型方法参数，WebSocket实现将把参数值传递到方法的此路径参数的值中。这样，就可以包括多个用@PathParam注解标记的String类型方法参数，假定每个@PathParam注解名称对应URI模板中的变量名。如代码清单6-4所示。

例如，可以通过如下代码重写 BookingService 端点。

代码清单 6-4：使用 URI 模板访问路径参数的注解式端点

```
@ServerEndpoint("/travel/{access-level}/flights")
public class BookingService {

  @OnOpen
  public void processNewClient(
      @PathParam("access-level") String memberLevel) {
    switch (memberLevel) {
      case "vip":
        this.offerBestSeatsAndPriorityDeals();
      default:
        this.offerCheapSeats();
    }

  }

  public void offerBestSeatsAndPriorityDeals() {}
  public void offerCheapSeats() {}

}
```

在该示例中，我们通过用@PathParam("access-level")注解 String memberLevel 方法参数来要求获取 access-level 变量。若发布端点的 URI 模板有更多的变量，可以通过简单地使用多个方法参数，每个参数通过一个合适的@PathParam 注解，以便在运行时从任意生命周期方法中获取它们。

使用@PathParam 时，方法参数的类型也有多种选择。如示例

中所示，最显而易见的和最常用的类型是 String，但是也可以使用 Java 基本类型或者其等价类。例如，当考虑路径参数的值是数字时，它可以使得代码中的工作更加方便和简洁。若如此，WebSocket 实现将尝试把入站 URI 匹配中产生的路径参数值解码为请求的类型。它将查找 Java 基本类型的带一个字符串参数的构造函数，并使用它来构造实例。如代码清单 6-5 所示的示例使用路径参数来指示由 WebSocket 支持的广告横幅中使用的酒店星级分类的级别。

代码清单 6-5：以整数值来访问路径参数

```
@ServerEndpoint("/travel/hotels/{stars}")
public class HotelBookingService {

  @OnMessage
  public void startAdvertisingBanner(String resortCode,
     @PathParam("stars") int hotelStars) {
    switch (hotelStars) {
      case 5:
        this.broadcastDeluxeProperties();
      case 4:
        ...
    }
  }
}
```

在该示例中，若客户端使用如下 URI

```
ws://fun.org/customer/services/travel/hotels/4
```

连接到服务，WebSocket 实现将解析出路径参数 stars 的值为 4。然后，它将传递此整数值 4 给 startAdvertisingBanner()方法。此整数值通过调用等价于下面代码的方法获得。

```
int hotelStars = (new Integer("4")).intValue();
```

当然，WebSocket 实现可能不能将客户端用于连接 WebSocket 的 URI 转换成注解式端点中生命周期方法其中之一请求的类型。若端点坚持要求路径参数是 String 类型，此时由于不需要转换，则这永远不是问题。然而，一旦选择以不同的 Java 类型接收路径参数，此转换有可能不可行。在该示例中，客户端可以使用如下 URI

```
ws://fun.org/customer/services/travel/hotels/oops
```

完美地连接上 HotelBookingService 端点并发送一条消息。

此时，路径参数的值是字符串 oops，然而消息处理方法看起来如下所示：

```
@OnMessage
public void startAdvertisingBanner(String resortCode,
        @PathParam("stars") int hotelStars)
```

WebSocket 实现被要求将字符串 oops 转换成整数，此转换当然无法完成。在这种情况下，以及所有这种转换失败的情况下，WebSocket 实现将不能调用消息处理方法，而是会产生一个错误(一个 DecodeException)。此错误将传递到端点上声明的错误处理方法，或者在没有定义错误处理方法的情况下记录日志，以便将来检查。此类转换错误不会导致连接关闭，然而，如果你决定关闭连接，可以在错误处理方法中关闭连接，并给出通知性的 CloseReason 以便客户端进行查看。

在我们从 Java WebSocket API 的 URI 模板主题离开之前，我们看看下一条规则，此规则用于处理 WebSocket 应用包含两个等价的 URI 模板时发生的情况。在 Java WebSocket API 中，只有两个 URI 模板拥有的路径变量在 uri-path 的同一位置，且非路径变

量的其他路径片段的值都相同，才认为它们等价。注意，两个 URI 模板的路径变量在 uri-path 的同一位置，即使它们在此变量片段使用不同的变量名称，也认为是等价的。例如，在 Java WebSocket API 中，如下两个 URI 模板被认为是等价的：

```
/{member-level}
/{access-code}
```

下面的 URI 模板

```
/booking/{transport-mode}/europe
/booking/{name}/europe
```

和

```
/a/{b}/c/{d}
/a/{x}/c/{y}
```

也是等价的。

现在看看规则 7，它提供了类似于规则 2(针对的是相等的 URI 路径)的规则(针对的是相等的 URI 模板)。

规则 7：两个拥有等价 URI 模板的端点不能部署在同一个 Web 应用中。

出现此规则的原因不难猜到：若 Java WebSocket API 在同一个 Web 应用中允许有映射到等价 URI 模板的多个端点的话，当客户端通过作为此 URI 模板的有效扩展的 URI 连接到 Web 应用时，不可能计算出客户端应该连接到哪个端点。

同规则 2 一样，若试图部署违反此规则的应用，则 WebSocket 实现将会部署失败。

6.2.4　在运行时访问路径信息

现在，我们已经介绍了发布端点到 WebSocket 实现的 URI 空间的两种机制，在查看一些基于这些机制构建的额外的选项之前，让我们在这里暂停片刻，查看在运行着的 WebSocket 服务器应用中与路径相关的一些 API。

首先，总是可以通过端点的 ServerEndpointConfig 对象，调用其

```
public String getPath()
```

方法获得 WebSocket 实现发布的端点的路径。此方法在端点映射成精确 URI 或者 URI 模板两种机制下都同样有效。示例如代码清单 6-6 所示：

代码清单 6-6：访问其自身路径映射的端点

```
@ServerEndpoint("/travel/hotels/{stars}")
public class HotelBookingService {

  public void handleConnection(Session s,
      EndpointConfig config) {
    String myPath = ((ServerEndpointConfig)
        config).getPath();
    // myPath is "/travel/hotels/{stars}"
  ...
}
```

你可能希望在运行时从端点内部访问的第二部分信息是客户端用来连接的 URI。下面你将看到此信息可以通过多种方式获得，但是包含所有信息的主要方法是：

```
Session.getRequestURI()
```

此方法提供了相对于 WebSocket 实现的 Web 服务器根的 URI 路径。注意，它包括包含 WebSocket 的 Web 应用的上下文根。例如，在酒店预定示例中，假设 HotelBookingService 端点部署的 Web 应用的上下文根是/customer/services。客户端通过 URI

```
ws://fun.org/customer/services/travel/hotels/3
```

连接到端点，则 HotelBookingService 端点从 Session 中调用 getRequestURI()方法获得的请求URI是

```
/customer/services/travel/hotels/3
```

当请求 URI 包含查询字符串时，Session 对象中还有两种方法用于进一步解析请求 URI 中的信息。因此，下面来看看查询字符串。

6.2.5 查询字符串和请求参数

正如在本章的开始部分所见，WebSocket 端点的 URI 路径是可选的查询字符串：

```
<websocket-protocol-scheme>://<authority>:<port-number>/
<uri-path>?<query-string>
```

URI 中的查询字符串最初是在 CGI(公共网关接口)应用中变得流行。URI 的 uri-path 部分可以定位 CGI 程序(经常是/cgi-bin)，uri-path 后追加的查询字符串能够为 CGI 程序提供一系列的参数来限定请求。查询字符串在 HTML 表单提交数据时也经常使用。例如，在 Web 应用的如下 HTML 代码中：

```
<form name="input" action="form-processor" method="post">
  Your Username: <input type="text" name="user">
                 <input type="submit" value="Submit">
```

```
</form>
```

当按下提交按钮时会产生一个 HTTP 请求，发给拥有此 HTML 代码的页面的相对 URI

```
/form-processor?user=Jared
```

其中，输入框包含文本 Jared。根据位于 URI 路径 /form-processor 的 Web 资源的性质，查询字符串 user=Jared 可用来确定应该构造何种类型的响应。例如，若位于 form-processor 的资源是 Java Servlet，则 Java Servlet 可以通过调用 API getQueryString()从 HttpServletRequest 中获取查询字符串。

按照类似的原则，查询字符串可以在用于连接使用 Java WebSocket API 创建的 WebSocket 端点的 URI 中使用。这马上引出了规则 8。

规则 8：查询字符串作为打开握手请求中的请求 URI 的一部分发送，它并不被 Java WebSocket API 用来判断它可能匹配的端点。

换句话说，不管请求 URI 中是否包含查询字符串，都不会对它匹配位于发布路径下的服务器端点产生影响。此外，查询字符串在用来发布端点的路径中也被忽略。

正如CGI程序和其他种类的Web组件所做的一样，WebSocket 端点可以使用查询字符串来进一步配置客户端的连接。因为 WebSocket 实现本质上会忽略入站请求中的查询字符串的值，所以任何使用此查询字符串的值的逻辑仅仅会在 WebSocket 组件内部。用于获取查询字符串的值的主要方法是 Session 对象的方法

```
public String getQueryString()
```

此方法返回所有的查询字符串(即?字符之后的所有字符)，以

及方法

```
public Map<String,List<String>> getRequestParameterMap()
```

它提供了从查询字符串中解析的所有请求参数的一个数据结构。你应该注意到映射的值是字符串列表；这是因为查询字符串可能有两个参数拥有相同的名称，但是其值却不同。例如，可以使用 URI

```
ws://fun.org/customer/services/
    travel/hotels/4?showpics=thumbnails&description=short
```

连接到 HotelBookingService 端点。在这种情况下，查询字符串是 showpics=thumbnails&description=short。为了从端点中获取请求参数，可以做一些事情，如代码清单 6-7 所示，其中 pictureType 和 textMode 的值分别为 thumbnails 和 short。

代码清单 6-7：访问请求参数

```
@ServerEndpoint("/travel/hotels/{stars}")
public class HotelBookingService {
  @OnOpen
  public void handleConnection(Session session,
        EndpointConfig config) {
    String pictureType =
      session.getRequestParameterMap().get("showpics").get(0);
    String textMode =
      session.getRequestParameterMap().get("description").get(0);
    ...
  }

  @OnMessage
  public void startAdvertisingBanner(String resortCode,
    @PathParam("stars") int hotelStars) {
  switch (hotelStars) {
```

```
    case 5:
      this.broadcastDeluxeProperties();
    case 4:
      ...
    }
  }
}
```

注意：

也可以从请求 URI 中获取查询字符串。在 Java WebSocket API 中，方法 Session.getRequestURI()的返回值总是包含 uri-path 和查询字符串。

既然已经探索过 URI 模板机制及其路径参数、查询字符串和其请求参数，你可能还想了解何时使用哪种机制。我们将在本章结束时回到此话题。

6.2.6　匹配优先级

许多 WebSocket 应用包含多个端点。因为有规则 2，所以你知道对于仅使用精确 URI 路径发布 WebSocket 端点的 WebSocket 应用来说，每个 WebSocket 端点必须有一个不同的值。然而，在 WebSocket 应用中也同样包含映射到 URI 模板的端点，因此可能存在两个端点路径同时匹配入站请求 URI 的情况。例如，假设一个 WebSocket 应用包含端点 ClementineEndpoint 和 CitrusEndpoint。这些端点映射如下：

```
ClementineEndpoint 映射到/clementine
CitrusEndpoint 映射到/{fruit}
```

显然，基于已有规则，客户端请求

```
ws://fruits.org/fruit-app/lemon
```

或者

```
ws://fruits.org/fruit-app/grapefruit
```

将都连接到 CitrusEndpoint。然而，对于如下 URI 来说情况又是如何？

```
ws://fruits.org/fruit-app/clementine
```

答案是使用此 URI 的客户端将连接到 ClementineEndpoint 端点，因为 Java WebSocket API 优先选择精确匹配。因为用来发布服务器端点的路径可能有多个片段，所以 Java WebSocket API 将入站 URI 与每个端点路径进行比较，从左至右读入 URI 路径的每个路径片段，在每个路径上精确匹配要优先于通过变量片段进行匹配。这就有了规则 9。

规则 9：在 WebSocket 应用中，当确定哪个 URI 路径和 URI 模板路径匹配入站 URI 时，Java WebSocket API 从左开始逐段比较路径片段，在每个片段上精确匹配要优先于通过路径变量进行匹配。

在我们继续介绍使用了此处一直查看的路径映射特性的应用之前，下面介绍这些映射优先级规则的示例。

假设有一个包含如下 3 个端点的应用：

- 端点 Apple，映射到/fruit/{plant-type}/thin
- 端点 Blackberry，映射到/fruit/bush/thin
- 端点 Clementine，映射到/fruit/{plant-type}/{skin-type}

注意，因为没有两个路径相等的情况，所有这些 URI 模板路径可以共存于同一 WebSocket 应用中。现在，若客户端连接到如下 URI：

```
ws://fruits.org/sweet-ones/fruit/bush/Thin
```

由于匹配是大小写敏感的(参见规则 3)，因此它将不会匹配任何端点。

若客户端连接到 URI：

```
ws://fruits.org/sweet-ones/fruit/bush/thin
```

则它将匹配 Blackberry 端点，但并不匹配 Apple 或者 Clementine 端点，因为精确匹配优先于 URI 模板的匹配。如果客户端连接 URI：

```
ws://fruits.org/sweet-ones/fruit/tree/thin
```

则它将得到与Apple端点的匹配，且其变量plant-type为tree，因为精确匹配片段优先于变量片段(规则9)规则排除了Clementine端点，即使它可能是匹配的。

最后，如果客户端通过入站 URI

```
ws://fruits.org/sweet-ones/fruit/evergreentree/thick
```

连接，则它将得到与 Clementine 端点的匹配，其路径参数 plant-type 的值是 evergreen，路径参数 skin-type 的值是 thick。原因在于入站 URI 是其映射的 URI 模板的一个有效扩展，而且它并不匹配其他两个端点。

最后，作为规则 9 的冗长阐述，假设一个上下文路径为 transport 的 Web 应用中有两个旅行端点。

- 端点 Plane 映射到 URI 模板：/{speed}/luxurious
- 端点 CruiseLiner 映射到 URI 模板：/slow/{comfort}

若客户端通过入站 URI

```
ws://fun.org/transport/slow/luxurious
```

连接到应用，则此 URI 匹配 CruiseLiner 且其路径参数 comfort 的

值是 luxurious，而不是匹配 Plane(其路径参数 speed 的值是 slow)。这是因为 Java WebSocket API 的精确匹配优于变量匹配的匹配优先级是从左到右计算的。

6.3 Portfolio 应用

为了让你知道何时使用精确 URI 路径映射以及何时使用 URI 模板，让我们转向 Portfolio 应用。Portfolio 应用是一个客户端/服务器 WebSocket 应用。客户端是一个简单的 JavaScript 客户端；服务器是使用 Java WebSocket API 创建的一个单一 WebSocket 端点。当客户端页面打开时，JavaScript 客户端连接到服务器端点，服务器端点依次广播一系列的股票投资组合更新返回给客户端。此应用的目标是能够很容易地在更大页面的某些部分嵌入正被频繁更新的数据。根据客户端用来连接服务器端点的 URI 的不同，此更新将包含或多或少的价格变化的细节信息和大量的股票投资组合的数据，并将以及时的或不及时的和有吸引力的方式传递。

下面介绍 Portfolio 应用的运行情况。

在页面的顶部是控制面板，它允许你来指定希望在更新中见到哪些股票数据字段，同时允许你选中你希望拥有的访问级别。控制面板的下面是输出区，它在每次服务器端点根据请求参数发送出股票组合的更新时进行更新。

你应该注意到在 Bronze 级别(参见图 6-1)，你的股票报价延迟了 20 分钟。当提升至 Silver 级别时，将得到一个新的配色方案，报价较新且更新更加频繁。在 Gold 级别时，报价能几乎无延迟地跟踪市场，并且有一个更吸引人的配色方案来反映提升后的身份。图 6-2 展示了 Gold 访问级别，它显示了所有的数据字段。

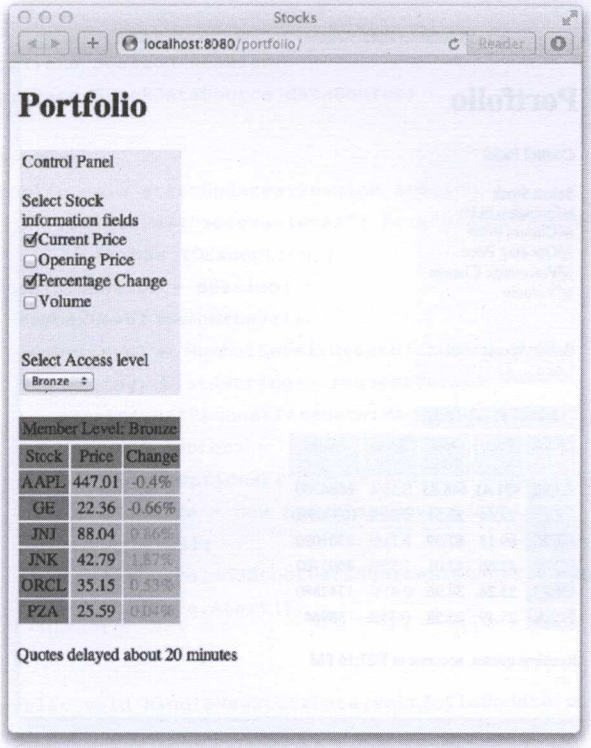


图 6-1　Bronze 级别的 Portfolio 应用

每次改变控制面板的选项时，JavaScript 客户端关闭现有的连接并重建新的连接。你希望看到的所有关于成员级别和数据字段的信息是由 JavaScript 每次用来连接的 URI 携带，并且事实上客户端从未给服务器发送一条 WebSocket 消息；客户端仅侦听来自服务器的更新消息。当客户端接收到来自服务器端点的更新消息时，它将其内容显示在 Web 页面上。

让我们看看此应用中最重要的一部分，也就是代码清单 6-8 所示的 PortfolioBroadcastEndpoint。

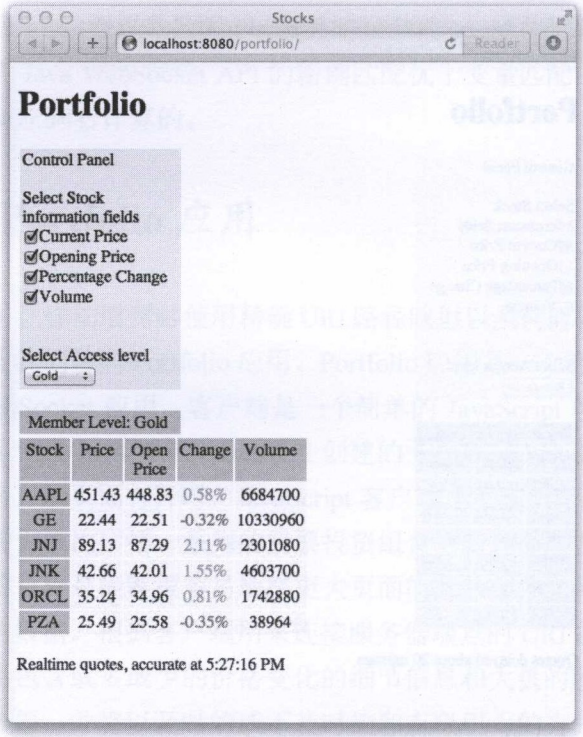


图 6-2 Gold 级别的 Portfolio 应用

代码清单 6-8：Portfolio 端点

```
import java.io.*;
import java.util.*;
import javax.websocket.*;
import javax.websocket.server.*;

@ServerEndpoint(
  value="/updates/{access-level}",
  encoders={PortfolioUpdateEncoder.class}
)
public class PortfolioBroadcastEndpoint implements
```

```
StockDataSourceListener {
  private Session session;
  private StockDataSource dataSource;

  @OnOpen
  public void startUpdates(Session session,
      @PathParam("access-level") String accessLevel)
          throws IOException {
    this.session = session;
    MemberLevel memberLevel;
    memberLevel = MemberLevel.create(accessLevel);
    Map<String, List<String>> requestParams =
      session.getRequestParameterMap();
    DataOptions options =
      this.parseOptionsFromRequestParams(requestParams);
    this.dataSource = new StockDataSource(options,
      memberLevel);
    this.dataSource.addStockDataSourceListener(this);
    this.dataSource.start();
  }

  public void handleNewStockData(PortfolioUpdate pu) {
    try {
      session.getBasicRemote().sendObject(pu);
    } catch (IOException | EncodeException ioe) {
      this.processError(ioe);
    }
  }

  @OnError
  public void processError(Throwable t) {
    System.out.println("Error: " + t.getMessage());
  }

  @OnClose
  public void stopUpdates(Session session) {
    dataSource.stop();
  }
```

```
    public DataOptions parseOptionsFromRequestParams(Map
        <String, List<String>> requestParams) {
      DataOptions options = (new DataOptions())
    .currentPrice(requestParams.containsKey(DataOptions.
        CURRENT_PRICE))
    .openPrice(requestParams.containsKey(DataOptions.
        OPEN_PRICE))
    .percentChange(requestParams.containsKey(DataOptions.
        PERCENTAGE_CHANGE))
    .volume(requestParams.containsKey(DataOptions.VOLUME));

      return options;
    }
}
```

首先需要注意的是 PortfolioBroadcastEndpoint 映射到 URI 模板路径/updates/{access-level}，而不是一个像本书中目前为止见到的所有端点一样的精确 URI。其次，注意 startUpdates()方法是此端点的打开事件处理程序，它有一个方法参数

```
@PathParam("access-level") String accessLevel
```

用于请求 access-level 路径参数的值。此值由 URI 模板路径匹配时产生，并传递给此方法。startUpdates()方法做的第一件事情是使用 accessLevel 变量的值创建 MemberLevel 对象。示例中此对象将用于请求股票数据源 StockDataSource 类的更新。通过浏览此应用其他的源代码可以看到 MemberLevel 类型是一个枚举类型，其值为 MemberLevel.BRONZE、MemberLevel.SILVER 和 MemberLevel.GOLD。startUpdates()做的第二件事情是基于客户端发送给连接的请求 URI 附带的查询字符串中携带的查询参数构造 DataOptions 对象。DataOptions 对象描述了将用于决定哪些字段显示在 Web 页面输出区的一系列的数据项。客户端基于 Web 页面中的控制面

板的哪些数据项选项被选中来制定查询字符串。注意示例中的查询字符串，请求参数中只有名称没有值，因为这些就是此示例所有所需要的。这意味着客户端用来连接的 URI 的种类与如下类似：

```
ws://localhost:8080/portfolio/
  updates/bronze?current-price&open-price&percentage-change
ws://localhost:8080/portfolio/
  updates/silver?current-price&percentage-change
ws://localhost:8080/portfolio/
  updates/gold?current-price&open-price&percentage-
    change&volume
```

PortfolioBroadcastEndpoint 类的 startUpdates() 方法使用 MemberLevel 和 DataOptions 对象来创建 StockDataSource 对象。StockDataSource类实现的细节和Java WebSocket API中的路径映射和处理没有特别的相关性，所以我们将不再详述它。此对象最相关的事情是当有新的股票信息更新时，它将回调 PortfolioBroadcastEndpoint。它通过一个称为PortfolioUpdate的对象进行回调。此PortfolioUpdate对象保存所有的股票信息。根据会员等级，StockDataSource将提供频繁的或者不频繁的更新。每当 PortfolioBroadcastEndpoint获得来自StockDataSource的回调(以调用其handleNewStockData()方法的形式)时，它立即把PortfolioUpdate对象发送回客户端。你应该从其@ServerEndpoint声明中注意到，PortfolioBroadcastEndpoint 使用了一个编码器。此 PortfolioUpdateEncoder 编码器接受股票信息、会员级别和 DataOptions，并格式化一个包含股票信息的HTML表格。此表格仅包含请求的数据项、适用的配色方案和数据新鲜度的声明。

另外，你将发现这种类型的端点按预定计划把更新广播给其客户端，并与客户端发送的任何消息都不相关。事实上，正如我们指出的，JavaScript客户端在这种情况下从来没有真正向

PortfolioBroadcastEndpoint端点发送消息。所有这些端点需要一个对Session对象的引用，它们一般会在打开处理方法中将Session对象存储为一个实例变量。

还请注意 PortfolioBroadcastEndpoint 从不处理任何显示相关代码。所有的发送回客户端的格式化 HTML 的代码都在 PortfolioUpdateEncoder 中。在 Web 应用程序中对于在哪里放置这样的显示代码有很多设计选项，但可以肯定的是把它独立于 PortfolioBroadcastEndpoint 中的消息处理逻辑之外总会是一个不错的选择。

一旦编码器编码了信息，WebSocket 实现就发送此 HTML 更新给 JavaScript 客户端来进行显示。并且这个循环直到来自于 StockDataSource 的下一个更新发出、客户端更改了控制面板的选项或者是用户离开了页面才结束。

6.4 查询字符串、路径参数与 WebSocket 消息

即使在这个简单的示例中，我们也面临着一些基本的设计选择，例如在何处存储定义了客户端需要的类型信息的各种参数，以及客户端如何接收它。

此应用选择使用查询字符串发送的请求参数来表示更新中发送的数据字段的种类，其本质上是定义了应该发送什么信息。它使用路径参数定义应该如何发送数据：会员级别路径属性是服务器端点用来决定多长时间发送更新和如何展示更新的关键。

然而，应用同样也可以定义 WebSocket 消息的格式。客户端将发送包含所有这些限定信息消息，帮助服务器端点构造会话。

那么应该选择在消息中编码这种类型的指定信息、采用路径参数集合形式还是采用在查询字符串中包含请求参数的集合形式？

与许多问题一样，没有一个真正明确的答案可以覆盖所有应用。然而，当你决定时可以考虑这里的 4 个指导方针。

指导方针 1：你想要发送的限定数据的结构如何？

如果你想要发送的限定数据的结构并非简单的键值对，那么将很难采用路径参数或请求参数形式。若信息的结构多于一个层级，你应该考虑创建一个消息格式使得可以通过 WebSocket 消息形式发送。尽管如此，若限定数据是一个简单的单值属性的列表，则通过让 Java WebSocket API 帮你解析出这些值可以大大节省消息编码解码代码的设计和实现的工作量。如果限定数据的属性是多值的，由于每个请求参数可以携带多个值，则此方法可能比只能用单值的路径参数更适合。

在 Portfolio 应用中，如果你想添加参数来管理当前价和开盘价的股票价格的小数点的位数，由于定义这样的参数将在查询字符串中需要携带的数据上引入另一个层级结构，因此此时你可能会决定是时候来创建一个消息格式来保存限定数据。

尽管如此，Portfolio 应用的这些数据能够很容易地用值为空的键值对来表示。这样它完全避免了必须定义携带数据的消息格式。应用能够简单地使用 Java WebSocket API 针对请求和路径参数的解析能力，而不必编写解码代码。

指导方针 2：你需要或者希望多长时间发送一次限定数据？

如果你的应用中有必要更新限定数据而不是每次关闭连接并重新建立一个新连接，将不能使用请求和路径参数，因为它们仅在连接第一次建立时被发送。

指导方针 3：数据量有多大？

虽然可以制定非常长的 URI，但是 Web 浏览器强加了其可以处理的 URI 的长度的上限。例如，Internet Explorer 能处理的 URI 的最大长度是 2 048 个字符。这应该是你决定是否使用路径参数或查询字符串这样的应用“元数据”的一个因素。

指导方针 4：应用中的其他路径映射是什么？

本章中你已经见过路径映射的一些规则，一旦应用中不止一个端点，这会对你如何组合路径映射有一些限制。若在打开连接时有简单扁平的限定信息要传送，路径映射规则可能会限制你广泛使用路径参数的选项。然而，由于查询字符串不是匹配入站 URI 到端点的处理决定的一部分，在应用中其他端点使用的查询字符串的格式完全不受限制。

6.5 WebSocket 路径映射 API 总结

在结束本章时，我们总结一下 Java WebSocket API 中与路径映射相关的 API，如表 6-1 所示。

表 6-1 WebSocket 路径映射 API

API	功 能
ServerEndpointConfig.Builder public static ServerEndpointConfig.Builder create(Class<?> endpointClass, String path)	创建时将路径值(相对 URI 或 URI 模板)指定给用于部署编程式端点的配置对象
@ServerEndpoint String value() attribute	用来定义注解式端点的路径值(相对URI或URI模板)的注解属性

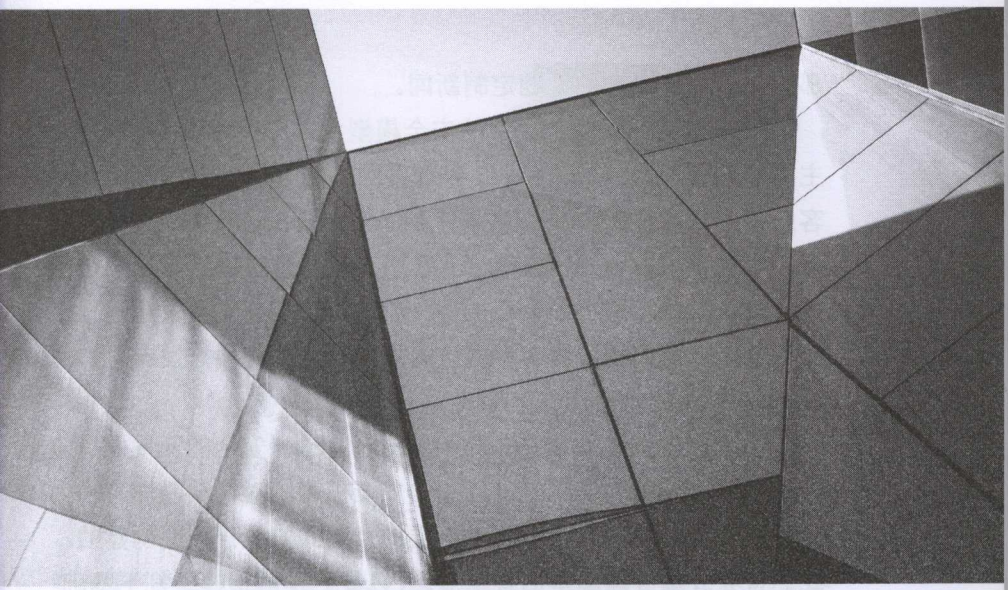
(续表)

API	功　能
ServerEndpointConfig public String getPath()	发布端点的路径(相对 URI 或 URI 模板)
Session public String getRequestURI()	客户端用来建立连接时相对于 Web 服务器根的 URI
Session public Map<String, String> getPathParameters()	当客户端连接到使用 URI 模板发布的端点时产生的键值对，其中键是 URI 模板变量的名称，值是客户端的请求 URI 中相应的路径片段
Session public String getQueryString()	客户端连接的请求 URI 的查询字符串部分
Session public Map<String, List<String>> getRequestParameterMap()	返回从客户端连接的请求 URI 的查询字符串中提取出的键-多值对的映射
@PathParam String value() attribute	方法参数级别的注解，可用于任一注解式端点的生命周期方法，用来提取当端点映射到 URI 模板路径时请求 URI 的路径参数。属性 value 指示要提取的路径参数的名称

6.6　本章小结

本章介绍了发布服务器 WebSocket 端点的所有选项。我们回顾了为注解式端点和编程式端点指定路径的机制，并查看了使用

精确 URI 和 URI 模板发布端点的两种机制。我们学习了 Java WebSocket API 中的 9 条路径映射规则，当客户端 WebSocket 端点尝试连接 WebSocket 服务器上发布的端点时，它们用于管理 Java WebSocket API 匹配入站 URI 的方式。本章最后给出了一个示例应用 Portfolio(该示例使得这些机制协同工作)，并讨论了在应用中应使用哪种映射技术的设计问题。



第7章

保护WebSocket服务器端点

你也许不希望所有人都能够访问你发布的 WebSocket 端点，并且对于有访问权限的人，你也许希望定制其交互来适应那个已经有访问权的人。如果你在社交网络上使用 WebSocket 广播你在地图上的位置，可能不希望包含你的位置的地图被任何人看到。如果一个 WebSocket 端点发送突发新闻更新，则你很可能希望为

那些你过去感兴趣的话题定制新闻。

Java WebSocket API 中的安全模型大部分关注于 WebSocket 主要的部署设置：一个运行在 Web 页面上的 JavaScript WebSocket 客户端访问运行在 Web 服务器上的 Java WebSocket API 服务器端点。本章介绍 Java WebSocket API 中有选择地控制特定用户访问 Java WebSocket 服务器端点的各种机制。我们将主要关注运行于服务器上以及被浏览器客户端访问的 WebSocket 端点发布时如何满足各种隐私约束。

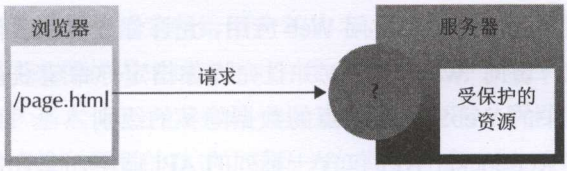
那些非常熟悉 Java EE 平台中 Java Servlet 安全的开发人员应该能够发现 Java WebSocket API 的安全模型相对简单易懂，因为它是基于 Java EE 平台中的 Web 层的声明式安全模型而构建的。我们首先介绍一些基本的安全概念，以便为理解该模型打下基础。

7.1 安全的概念

下面先从基本部署场景开始介绍：某人使用 Web 浏览器访问 Web 服务器上的资源(参见图 7-1)。

此图中，用户正在初始化一个来自 Web 页面的请求，该页面被载入由服务器上的安全模型所保护的资源的浏览器中。在请求被满足前，服务器的安全模型必须考虑如下 3 个问题。

- 谁正在请求资源？如果用户没有登录到服务器，则此请求是匿名的。这可能是因为安全模型允许匿名访问资源，或者可能是服务器回答下一个请求时必须知道请求资源的用户的身份。



- 谁正在请求资源?
- 这个人可以访问它吗?
- 数据的隐私性如何?

图 7-1　3 个安全问题

- 这个人可以访问它吗？如果服务器不允许匿名访问资源，那么它必须在某个地方保存必要的信息来决定是否允许特定的用户访问此资源。一旦服务器作出决定，它将需要考虑下一个问题。
- 数据的隐私性如何？服务器响应请求时可能已经决定此资源产生的数据必须被传递回客户端，且必须以一定的程度保证信息途中不会被改变。或者服务器可能希望保证，在一定程度下，任何拦截从资源发送给客户端数据的人都无法读取这些数据。

确定这些问题的答案的过程通常包括如下：

- **认证**　用户发出请求的身份的通信过程
- **授权**　允许或者拒绝访问资源的过程
- **保证数据隐私**　决定如何保证资源和客户端用来传输数据的通信信道的隐私性

7.2　Java WebSocket API 安全

Java WebSocket API 包含一系列的特性，允许为包含你希望保护的 WebSocket 端点的 Web 应用配置安全模型。它允许你配置

各种各样的认证方案来访问 Web 应用，允许你授权或者拒绝特定类型的用户访问 Web 应用，并且允许你指定你希望获得的访问 Web 应用中的 WebSocket 端点的数据隐私的级别。

Java WebSocket API 包含一系列的 API 调用，允许你从运行的应用中访问安全模型的多个方面，同时也允许你在 Java WebSocket API 内置的特性不满足应用需求时将自定义的特定于应用的安全特性构建到应用中。

当然，除了我们刚刚描述的 3 个核心特性之外，每个优秀的安全模型还有更多的功能，例如审核、不可抵赖性和互操作性。Java WebSocket API 的每个实现都对这些特性有某种程度的支持，但是不同的实现支持程度可能有所差异。我们将关注这 3 个核心特性，因为 Java WebSocket API 以标准方式支持它们，并且它们覆盖了 Java WebSocket 应用在安全方面的主要需求。

下面先介绍在 Java WebSocket 应用中如何配置身份认证。

7.2.1 认证

如果你打算只允许通过身份认证的用户使用你的 WebSocket 端点，那么当 Java WebSocket API 运行在 Java EE 实现中时，会有一些方案供你选择，这一点你将在下面的子小节中看到。

1. 基本认证

这是最简单的基于 HTTP 的认证方案。客户端发送 HTTP 头中已编码的用户名和密码对(经常被称为凭据)。客户端可能预先发送此凭据，或者是在客户端发送一个未经身份认证的请求时对服务器发出的认证挑战进行响应。在 WebSocket 应用中，如果你需要一种类型的登录并且想知道哪个用户正在访问 WebSocket，这是一个可用的良好方案。在应用中，因为你已经知道关于用户

的信息，因此当希望定制给用户的内容，但是不一定需要一个强的安全模型来保护发送或者接收的数据时，这种方案十分有用。因为 HTTP 基本认证方案中的凭据发送时未加密，所以它们非常容易受到拦截和解码的攻击。正因为如此，许多 Web 应用希望选择仅在加密的或者私有的连接上使用 HTTP 基本认证来保护凭据和应用数据。

一般情况下，浏览器使用用户名密码的基本模态对话框来获取用户凭据，因此其用户体验虽然简单，但不易定制。

2. 基于表单的认证

基于表单的认证是指一种定义了一系列的请求参数来表示用户名和密码的机制，并且它被编码在 Web 页面的一个特殊的<form>元素中。同基本认证一样，凭据发送时未加密，所以它适合于其同类的应用。基于表单的认证和 HTTP 基本认证的主要区别是登录表单的外观界面和布局是可定制的，所以它适合更关注用户体验的应用。

3. 摘要式认证

摘要式认证这种机制从客户端收集一个用户名和密码的组合，并以受保护的形式发送给服务器用来验证。因为凭据以受保护的形式发送，所以它本质上是比 HTTP 基本认证和基于表单的认证更安全的一种认证形式。然而，由于客户端无法知道它是将凭据信息发送到正确的服务器而不是伪装的服务器，因此它不是最安全的加密方式。

4. 客户端证书认证

客户端证书认证是服务器和可选的客户端使用数字证书相互

认证的一个过程，其中数字证书作为一种加密的手段来验证代理的身份。此过程本质上比 HTTP 基本认证或基于表单的认证更安全，因为客户端能够验证服务器的身份，而凭据经常通过 SSL 使用 HTTPS 传输。

遗憾的是，当通过 JavaScript WebSocket 端点直接尝试未认证的握手时，这些机制中没有一个机制能够被可靠地调用。虽然理论上 WebSocket 协议允许其实现在打开阶段握手时插入其自有的认证交互，但 WebSocket 协议的实现通常不这么做。尽管如此，当浏览器尝试来自于 Web 页面中的 JavaScript WebSocket 的打开阶段握手时，其发出的打开阶段握手携带了此封闭的 Web 页面中的认证信息。这意味着，为了访问一个需要从客户端 JavaScript WebSocket 进行认证的 WebSocket 端点，你必须确保包含此 JavaScript WebSocket 的 Web 页面已经对可授权访问端点的一个用户身份进行了认证。

作为为应用选择身份认证方案的动机，如果你希望限制所有 WebSocket 端点或者是其子集被已知用户访问，或者是应用中的 WebSocket 端点需要在运行时访问用户身份(可能是做某种应用级别的安全工作，或者是简单地显示用户名称)，将需要对用户进行认证。

一旦你决定好希望使用的方案，其配置相对简单：在打包端点的 WAR 文件的部署描述符中指定即可。部署描述符中包含的关键元素是<login-config>元素。表 7-1 列出了其子元素。

表 7-1 <login-config>元素的子元素

元素名称	值	目 的
auth-method	BASIC、FORM、DIGEST 或者 CLIENT-CERT	定义认证方案

(续表)

元 素 名 称	值	目　的
realm-name	基本认证中使用的用户域的名称	定义哪个服务器域将用来验证基本认证凭据
form-login-config	子元素login-page和error-page	仅用于基于表单的认证，定义了 Web 应用中包含登录表单的页面的相对路径，以及登录失败时用户被重定向到的页面

示例

这里有 web.xml 文件的一些片段，它们指定了在 WebSocket 应用中可以使用的 4 种类型的认证方式。

对于代码清单7-1所示的基本认证来说，web.xml顶层web-app元素包含的如下XML片段将配置应用在尝试访问一个受保护资源时来接收一个HTTP基本认证的挑战。服务器将通过其file认证域匹配凭据来验证客户端发送的凭据。

代码清单 7-1：基本认证

```
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>file</realm-name>
</login-config>
```

在代码清单 7-2 所示的示例中，将看到如何指定你的应用使用表单认证。如果 web.xml 的顶层 web-app 元素中包含如下 XML，服务器会将对 Web 应用中受保护资源的任何未认证的请求重定向到位于 Web 应用的 URI 名称空间的根下的 login.html 页面。

代码清单 7-2：表单登录认证

```
<login-config>
  <auth-method>FORM</auth-method>
  <form-login-config>
  <form-login-page>/login.html</form-login-page>
  <form-error-page>/error.html</form-error-page>
  </form-login-config>
</login-config>
```

现在，如果 login.html 页面包含代码清单 7-3 所示的示例中的 HTML 表单，那么提交表单时就会将名为j_username 和j_password 的请求参数(包含用户输入的用户名和密码的值)提交给服务器上一个名为 j_security_check 的专门保留的资源(用于处理认证请求)。如果认证成功，用户将被重定向到其起初请求的受保护的资源。如果认证失败，用户将被重定向到 error.html 页面。此页面在 form-login-config 元素中指定，位于 Web 应用的 URI 空间的根路径下。

代码清单 7-3：一个登录表单<form>

```
<form method=POST action="j_security_check">
   User ID
  <input type="text" size="10" name="j_username"> <br>
   Password
  <input type="password" size="10" name="j_password">
  <input type="submit" name="login" value="Login">
</form>
```

为了指定应用使用摘要式认证，可在 web.xml 部署描述符文件中的顶层 web-app 元素下包含代码清单 7-4 所示的 XML 代码。

代码清单 7-4：摘要式认证

```
<login-config>
  <auth-method>DIGEST</auth-method>
</login-config>
```

这意味着每当用户试图访问应用中的一个受保护的资源时，他将面临摘要式认证的挑战。

最后，如果你想要在 Web 应用的未认证用户尝试访问受保护的资源时服务器初始化一个客户端证书的挑战，则需要在 Web 应用中包含 login-config 元素，如代码清单 7-5 所示。

代码清单 7-5：客户端证书认证

```
<login-config>
  <auth-method>CLIENT-CERT</auth-method>
</login-config>
```

如果使用此认证方式，应该需要确保已经安装了一个已验证的客户端证书，服务器将信任并且认可该证书。

除非你首先知道如何保护那些服务器端点，否则为包含希望访问服务器端 Java WebSocket 端点的客户端 JavaScript WebSocket 端点的 Web 页面指定认证方案并没有什么用。因此，我们转而介绍 Java WebSocket 应用中的认证语言。

7.2.2　授权

授权是 WebSocket 实现决定一个特定用户是否允许访问一个特定端点的过程。Java EE 平台使用一个称为用户角色(或者简称为角色)的间接层来定义用户。角色是用户的一种抽象，在无须开发人员在应用配置中硬连接实际用户名的情况下，允许在应用中建立授权规则。

当一个已认证的打开阶段握手请求到达服务器时，为了决定是否应该授权对端点的访问，服务器必须执行 3 件事情。

首先，服务器必须确定已认证用户所属的角色或者角色组。此信息并不是 Java WebSocket API 定义的标准配置中的一部分，所以其对于不同的 WebSocket 实现来说将不同。在 Glassfish 4.0 应用服务器中，可以在针对 Glassfish 的名为 glassfish.xml 文件的部署描述符中构造用户和角色间的关联，并将其与端点一起打包到 WAR 文件中，但是其他应用服务器会使用不同的方案。

其次，一旦服务器知道打开阶段握手请求所属的角色，它就通过应用中的 Web 部署描述符(同 WAR 文件一起打包的 web.xml 文件)进行判断。下面你将看到一些特定的示例，但是目前知道对于一个给定的 URI(打开握手阶段的 URI)，web.xml 文件中的安全约束条件包含了所有必要的信息来判断哪些角色可以访问该 URI 就足够了。

最后，如你所见，WebSocket 端点以其自有的配置机制映射到一个路径，编程式端点可以使用 ServerEndpointConfig，注解式端点可以使用 ServerEndpoint 注解中的值属性。

通过这 3 条信息，对于一个给定的用户和打开阶段握手请求的 URI，服务器就可以决定是否允许用户所属的所有角色访问此 URI。

此机制及做出决定所需的配置信息的位置如图 7-2 所示。

下面介绍 web.xml 中限制一组用户角色访问端点的配置语法。

此语法使用的元素是 security-constraint。为了定义希望限制访问的 URI，需要定义一个子元素 web-resource-collection，此子元素包含如下子元素(参见表 7-2)：

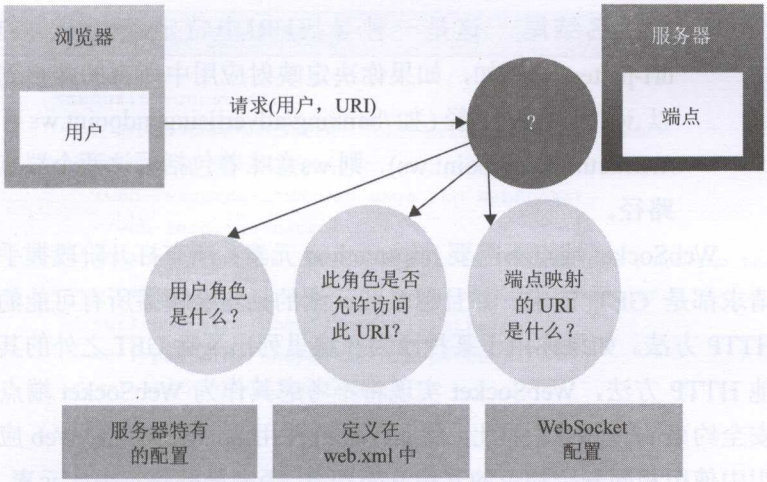


图 7-2　授权决策

表 7-2　web-resource-collection 子元素

web-resource-collection 子元素	多重性	值
web-resource-collection-name	单值	显示在工具中的 URI 集合的可选文本名称
description	单值	显示在工具中的 URI 集合的可选文本描述
url-pattern	多值	指示一个 URI 或一组 URI 的一个路径
http-method	多值	用来访问资源的 HTTP 方法(GET、POST 等)

可以通过下列方式之一指定 url-pattern 元素中的路径。

- **精确路径**　此路径相对于 Web 应用的上下文根——例如 /websockets/myendpoint。
- **通配符**　此路径定义了相对于 Web 应用的上下文根的一组 URL。例如，/websockets/*将指示所有以 websockets 开头的 URI。

- **文件名结尾** 这是一种寻找URI中特定端点模式的url-pattern。例如，如果你决定映射应用中所有的端点到以.ws结尾的路径(如/banking/advertisingendpoint.ws、/marketnewsendpoint.ws)，则.ws意味着包括了这两个端点路径。

WebSocket端点不需要http-method元素；所有打开阶段握手请求都是GET请求，并且忽略此元素的话将会覆盖所有可能的HTTP方法。如果你由于某种原因在这里列出了除GET之外的其他HTTP方法，WebSocket实现将不考虑其作为WebSocket端点安全约束的一部分。因此，除非你额外使用此语法来保护Web应用中使用相同安全约束的其他Web组件，否则最好是省去此元素。

一旦在安全约束中建立了一个或者多个web-resource-collection，就需要指出哪些角色允许访问web-resource-collection中指定的URI。为此，简单地为security-constraint添加一个auth-constraint元素列表即可，其中的每个元素包含你希望授权访问的角色名称。此处列出的任何角色名称都必须已在web.xml的security-role元素中声明。

示例

让我们看一些示例。假设我们有一个包含一个Web页面和一个服务器端点的Web应用。Web页面名为home.html，端点映射到/endpoint。Web页面home.html包含一个将访问服务器端点的JavaScript客户端WebSocket端点。我们希望仅允许那些在customer角色中的用户能够访问Web页面和端点。因为此Web应用并不要求最严格的安全策略，所以我们很高兴地使用基本认证方式来认证用户。代码清单7-6展示了需要此策略保护Web应用的web.xml文件中的安全片段。

代码清单 7-6：安全约束示例

```
<security-constraint>
  <display-name>WebSocket Constraint</display-name>
  <web-resource-collection>
    <web-resource-name>web page and WebSocket
    </web-resource-name>
    <description>This restricts access to the web page and the
       WebSocket it will try to access</description>
    <url-pattern>/home.jsp</url-pattern>
    <url-pattern>/endpoint</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <description/>
    <role-name>customer</role-name>
  </auth-constraint>
</security-constraint>
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>file</realm-name>
</login-config>
<security-role>
  <role-name>customer</role-name>
</security-role>
<security-role>
  <role-name>administrator</role-name>
</security-role>
```

现在，当新用户尝试访问 home.html 页面时，由于 home.html 页面仅允许那些角色为 customer 的用户来访问，如前述示例中 security-constraint 所示，因此服务器将发出一个 HTTP 基本认证挑战。注意此 Web 应用有两个角色：customer 和 administrator。如果挑战获得成功，则用户能够访问 home.html 页面。如果此用户导致 home.html 页面尝试通过 Web 页面中的 JavaScript 客户端端点连接映射到/endpoint 的 WebSocket 服务器端点，服务器将拦

截代表客户端端点的浏览器发出的打开阶段握手。此打开阶段握手请求将携带用户的认证上下文(除非用户在此期间已经登出，或者是会话超时)。服务器将使用此上下文来验证对/endpoint URI 的访问。如果打开阶段握手携带正确的认证上下文，则打开阶段握手将继续并会建立连接。

注意，通过客户端以任何方式发出的到此受保护的 WebSocket 服务器端点的未认证的打开阶段握手都将失败。特别是，任何未经认证的打开阶段握手将收到一个 HTTP 401(未授权)响应而不是成功的打开阶段握手响应。

通过图 7-3 可以看到一系列步骤，可按照从上到下的顺序来进行阅读。

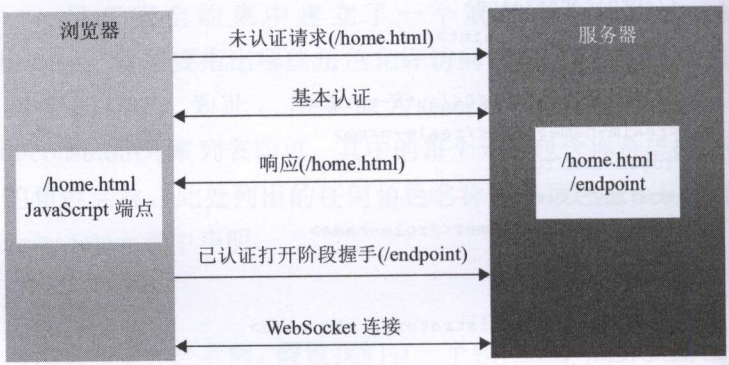


图 7-3 授权交互过程

7.2.3 私有通信

为了保护 WebSocket 端点间的数据交换，web.xml 部署描述符中的语法允许你标记 URI 空间中的一部分为如下之一：

- INTEGRAL 这意味着当在传输过程中改变数据时，没有第三方能够篡改数据。

- CONFIDENTIAL　这意味没有第三方能够在交换过程中篡改或者甚至是读取数据。

多数应用服务器把任一选项解释成它们仅允许使用一个安全连接(也就是通过SSL的WebSocket，或称为wss)访问资源(在这里就是服务器上的WebSocket端点)。为了确保JavaScript WebSocket和使用Java WebSocket API开发的服务器端WebSocket间的私有通信，有两件事情需要指定。

首先，客户端必须在其打开阶段握手中指定wss://，并且使用托管WebSocket的Web服务器的SSL端口。下面是通过JavaScript代码创建使用wss://进行连接的WebSocket对象的一个简单示例：

```
var myWebSocket = new WebSocket("wss://localhost:8181/
    SecureApp/secure-endpoint")
```

此示例中，服务器使用端口 8181 来进行 SSL 连接。

其次，包含服务器 WebSocket 端点的 Web 应用需要指定它仅能够通过私有的 WebSocket 连接访问。为此，必须定义一个 security-constraint，其中的 url-pattern 用于匹配你希望进行私有通信的端点的路径，并且为其添加一个 user-data-constraint。表 7-3 展示了 user-data-constraint 的子元素。

表 7-3　user-data-constraint 的子元素

user-data-constraint 子元素	值	目　的
description	文本	当通过工具查看部署描述符时约束的可选文本描述

(续表)

user-data-constraint 子元素	值	目　的
transport-guarantee	NONE、INTEGRAL 或 CONFIDENTIAL	约束定义的隐私级别。默认值 NONE 表示不需要私有通信；INTEGRAL 和 CONFIDENTIAL 仅能通过 wss 访问

例如，可以通过将如下的 user-data-constraint 添加到 security-constraint 中来调整上述示例，从而说明其授权。这意味着页面和其中的端点仅可被 customer 角色中的已认证用户通过 wss 而不是 ws 来访问。

因此，现在你需要调整 JavaScript WebSocket 使用的 URL，使用 wss 代替 ws 来确保连接的两端都同意此连接是私有的。如代码清单 7-7 所示。

代码清单 7-7：CONFIDENTIAL 连接

```
<user-data-constraint>
  <transport-guarantee>CONFIDENTIAL</transport-guarantee>
</user-data-constraint>
```

如果你希望在 Web 页面到端点的通信中使用 wss，一般情况下，你将仅能够在已经同 Web 应用建立 HTTPS 连接的浏览器中这样做。这意味着实际上如果你有一个包含 JavaScript WebSocket 端点的 Web 页面，希望使用 wss 来连接服务器端的 Java WebSocket 端点，则需要确保对 JavaScript 端点和拥有它的 Web 页面存在一个 security-constraint，其中它的 user-data-constraint 被设置成 INTEGRAL 或者 CONFIDENTIAL。这与你已在认证中遇到的情

景类似；如果你希望访问一个需要认证的服务器端 Java WebSocket，则持有客户端 WebSocket 的 Web 页面必须已被认证。

7.2.4　Java WebSocket 安全 API

在运行时可以访问多种 API，它们为你的应用提供了一个视图，可以了解你在 Web 应用部署描述符中设置的安全模型。

1. 会话对象中可用的安全信息

访问连接到 WebSocket 的已认证的用户通常是很有用的。此信息可从 Session 对象中通过如下方法调用来获得。

```
public Principal getUserPrincipal()
```

其返回值是表示用户的一个 java.security.Principal 对象，你可以从中获取名称，并且通过进一步检查其实际的子类型，可以获取关于用户及其关联的凭据的更多信息。如果连接未被认证，此方法返回空。此方法在很多方面都有用，从在端点要发送的 WebSocket 消息中嵌入用户名称，到在应用中构建自定义的授权检查，而不再依赖于本章前面介绍的 Web 容器中已经提供的安全约束机制。

从 Session 对象中也可以确定连接是否是私有的，也就是说，WebSocket 连接是否是加密的。此信息可通过使用如下方法调用来获得：

```
public boolean isSecure()
```

2. HandshakeRequest 中可用的安全信息

正如第 4 章所述，可以拦截服务器 WebSocket 端点中的打开阶段握手请求。为此，通过子类化 ServerEndpointConfig.Configurator

类并覆盖其

```
public void modifyHandshake(ServerEndpointConfig sec,
    HandshakeRequest request, HandshakeResponse response)
```

方法，同时通过链接此端点到此配置器类(对于注解式端点来说，使用@ServerEndpoint 的 configurator 属性；当部署编程式端点时，则是设置 ServerEndpointConfig 实例的配置)来实现。

若如此，可以通过 modifyHandshake() 方法访问 HandshakeRequest对象。这与你已经在Session对象中看到的方法一样：

```
public Principal getUserPrincipal()
```

另外，HandshakeRequest 还有下列方法：

```
public boolean isUserInRole(String rolename)
```

此方法旨在允许你确定构造打开阶段握手请求的已认证的用户是否是传入的角色名称的一个成员。例如，如果你在包含端点的 Web 应用的 web.xml 部署描述符中列出一个名为 customer 的 security-role，当从客户端尝试连接到此端点时，可从获得的 HandshakeRequest 对象确定客户端的用户身份是否是 customer 角色的成员。如果你希望根据活动用户所属的角色在端点中提供不同等级的交互，或者如果你希望提供自定义的安全模型作为应用的一部分而不是依赖于 Web 容器中提供的安全约束机制，这都是有用的。如果客户端构造一个未认证的连接到端点，那么此方法一直返回 false，就如同传入一个 Web 应用不知道的角色一样。

本章的示例应用将使用这些安全 API，但是在离开本话题之前，表 7-4 给出了 Java WebSocket API 中所有与安全相关的 API 的摘要。

表 7-4　安全 API

API 对象	API 调用	目　的
Session	Principal getUserPrincipal()	获取连接的已认证用户
Session	boolean isSecure()	确认当前建立的连接是否加密
HandshakeRequest	boolean isUserInRole(String rolename)	确认当前已认证用户是否属于提供的角色
HandshakeRequest	Principal getUserPrincipal()	获取尝试打开连接的已认证用户

7.3　Stock Account 应用

我们将通过一个示例完成对 Java API WebSocket 安全模型的介绍。Stock Account 应用是一个通过浏览器客户端访问的 Web 应用。在其主入口页面，你应该会发现一个横幅，其中包含关于一些公司的价格和股票的市场信息，如图 7-4 所示。

当你登录时，被要求提供用户名和密码，之后被带到一个账户界面。如图 7-5 所示，你仍然可以看到一个横幅，但是其中的财务信息更及时并且更新更频繁。此外，可以看到股票投资组合的当前持仓的一个清单。也将看到当你拥有的股票的新的市场数据可用时，你的股票投资组合的现行市价正不断更新。你可以从此页面登出应用，并返回到原来的主页面。

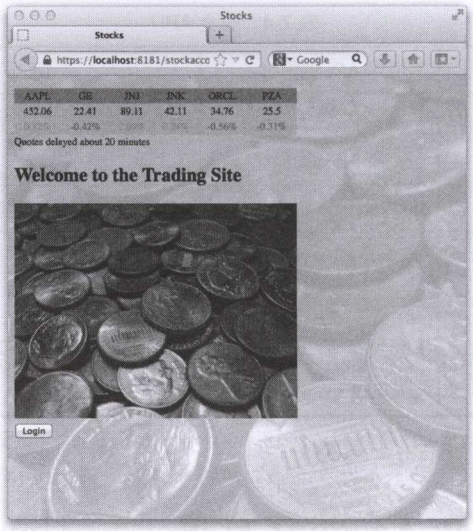


图 7-4　进入 Stock Account 应用

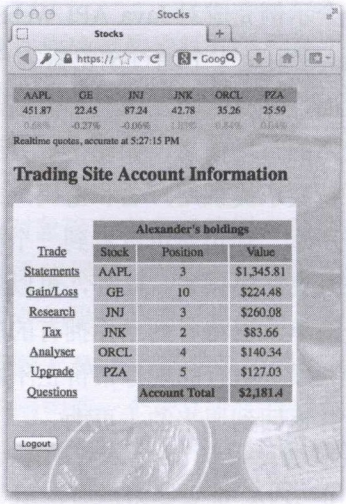


图 7-5　金牌用户的账户信息

注意，入口页和账户页都只能通过 HTTPS 访问。当然同时也请注意到为了访问账户页，必须提供一个用户名和密码。此应用预先配置两个用户：Alexander 和 Jesse，运行于 Glassfish 4.0 应用服务器上。你应该注意到，根据是以 Alexander 身份登录还是以 Jesse 身份登录，你将获得不同级别的服务。

应用内部有两个服务器端端点，其类名称和路径映射如表 7-5 所示：

表 7-5　端点类及路径映射

端　点　类	URI 映 射
AccountEndpoint	/account/info
PortfolioBroadcastEndpoint	/banner/{secure-access}

你应该已经对 PortfolioBroadcastEndpoint 多少有些熟悉。它基于上一章介绍过的 Portfolio 示例中的同名的端点。其现在的目的实际上是发送新股票价格的通知，作为一种股票行情功能。当应用中的 MemberLevel 越高时，更新会更频繁，数据更及时。你在后面将马上看到如何决定会员级别。AccountEndpoint 具有更新账户信息的能力，它获取当前市场数据并且基于用户每只股票的当前持仓情况来计算用户持有的每只股票的现行市价。此外，AccountEndpoint 使用 MemberLevel 来提供不同的服务级别。

代码清单 7-8 介绍此应用的部署描述符。

代码清单 7-8：Stock Account 应用的部署描述符

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/
javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/
ns/javaee http://java.sun.com/xml/
```

```
ns/javaee/web-app_3_0.xsd">
<session-config>
  <session-timeout>
    30
  </session-timeout>
</session-config>
<security-constraint>
  <display-name> login access</display-name>
  <web-resource-collection>
    <web-resource-name>Account Info</web-resource-name>
    <url-pattern>/index.jsp</url-pattern>
  </web-resource-collection>
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
<security-constraint>
  <display-name>customer_access</display-name>
    <web-resource-collection>
      <web-resource-name>Account Info</web-resource-name>
      <url-pattern>/account.jsp</url-pattern>
      <url-pattern>/account/info</url-pattern>
      <url-pattern>/banner/secure</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>customer</role-name>
      <role-name>premium_customer</role-name>
    </auth-constraint>
    <user-data-constraint>
      <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
  </security-constraint>
  <login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>file</realm-name>
  </login-config>
  <security-role>
    <role-name>customer</role-name>
```

```
    </security-role>
    <security-role>
      <role-name>premium_customer</role-name>
    </security-role>
  </web-app>
```

首先需要注意的事情是，Stock Account 应用使用两个安全约束来限制访问 Web 页面和两个端点。文件 web.xml 中的第一个安全约束为访问 index.jsp 首页设置了一个私有通信的条件。它确保你仅能够使用 HTTPS 访问 index.jsp。第二个安全约束为 account.jsp 和 AccountEndpoint 端点设置了一个授权控制。它仅允许那些角色为 customer 或 premium_customer 的用户访问这些资源。你应该也看到同样的安全约束限制了对 URI /banner/secure 的访问(潜在地匹配 PortfolioBroadcastEndpoint 端点)为那些有同样角色的用户。这确保了试图访问账户页、AccountEndpoint 端点或者通过路径/banner/secure 访问 PortfolioBroadcastEndpoint 的任何用户都必须被认证并且属于正确的角色。下面查看 web.xml 中的 login-config 元素，你应该看到认证方法是 HTTP 基本认证，然而由于试图打开 account.jsp 而引起登录的 index.jsp 页面本身只可以通过 HTTPS 访问，因此 HTTP 基本认证信息在加密连接上传输。除非你非常宽松地对待身份认证，否则这通常是一个要养成的好习惯。

你应该注意到当入口页index.jsp访问PortfolioBroadcastEndpoint时，它使用的URI是/banner/nouser而不是账户页所使用的/banner/secure。当然，由于index.jsp上面没有放置认证控制(仅仅是user-data-constraint)，因此这意味着你不必登录就能够看到页面顶部的股票行情信息。

为了计算出每个用户的 MemberLevel，每个端点使用一个自定义的配置器。MemberLevel 用于应用的关键部分，以便决定服

务级别和账户页面的显示方案。让我们看看这些是如何完成的。在 MemberLevelConfigurator 类中，你将看到我们已经重写了 modifyHandshake()方法，如代码清单 7-9 所示。

代码清单 7-9：MemberLevelConfigurator

```
public class MemberLevelConfigurator extends
      ServerEndpointConfig.Configurator {
public static String CUSTOMER = "customer";
public static String PREMIUM_CUSTOMER = "premium_customer";
public static String MEMBER_LEVEL = "MemberLevel";

  public void modifyHandshake(ServerEndpointConfig sec,
      HandshakeRequest request,
      HandshakeResponse response) {
    MemberLevel ml = null;
    if (request.isUserInRole(CUSTOMER)) {
      ml = MemberLevel.SILVER;
    } else if (request.isUserInRole(PREMIUM_CUSTOMER)) {
      ml = MemberLevel.GOLD;
    } else {
      ml = MemberLevel.BRONZE;
    }
    sec.getUserProperties().put(MEMBER_LEVEL, ml);
  }
}
```

从代码清单中可以看到，此方法实现时使用当前用户的用户角色来确定访问级别。用户 Alexander 是 premier_customer 角色的成员，所以他将被授予金牌会员身份，如你之前所见。用户 Jesse 是 customer 角色的成员但不是 premier_customer 角色的成员，所以他支付的服务费用较低，他将被授予银牌会员身份，如图 7-6 所示。

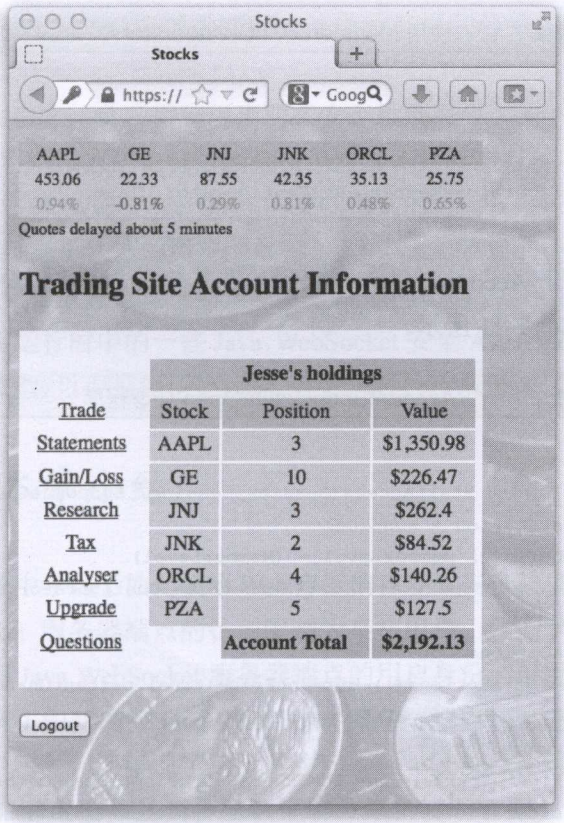


图 7-6　银牌用户的账户信息

在 MemberLevelConfigurator 中，modifyHandshake()方法实现将 MemberLevel 插入 ServerEndpointConfig 对象中。

两个端点都使用 MemberLevelConfigurator，如类级别的 WebSocket 注解中所示。例如，对于 AccountEndpoint，用法如代码清单 7-10 所示。

代码清单 7-10：使用 MemberLevelConfigurator

```
@ServerEndpoint(
  value="/account/info",
  encoders={AccountUpdateEncoder.class},
  configurator=MemberLevelConfigurator.class
)
```

而对于 AccountEndpoint 来说，在其@OnOpen 方法中的用法如代码清单 7-11 所示。

代码清单 7-11：AccountEndpointOnOpen 处理程序

```
@OnOpen
public void startAccess(Session session, EndpointConfig ec)
  this.session = session;
  MemberLevel memberLevel = (MemberLevel)
ec.getUserProperties().get(MemberLevelConfigurator.MEMBER_
    LEVEL);
  DataOptions options = (
    new DataOptions()).currentPrice(true).percentChange(true);
  this.account = new Account(session.getUserPrincipal(),
     memberLevel);
  this.dataSource = new StockDataSource(options, memberLevel);
  this.dataSource.addStockDataSourceListener(this);
  this.dataSource.start();
}
```

很容易看到，通过 MemberLevelConfigurator 设置的 MemberLevel在此被检索出来，以便建立来自驱动市场数据更新的数据源的服务级别。

类似地，PortfolioBroadcastEndpoint使用MemberLevelConfigurator从已认证的账户信息页面确定其MemberLevel。如果你通过未认证的连接(事实上是入口页index.jsp)访问PortfolioBroadcastEndpoint，那么用户Principal对象为null，并且MemberLevelConfigurator中使

用的isUserInRole()一直返回false，最后将得到最低的铜牌服务级别。

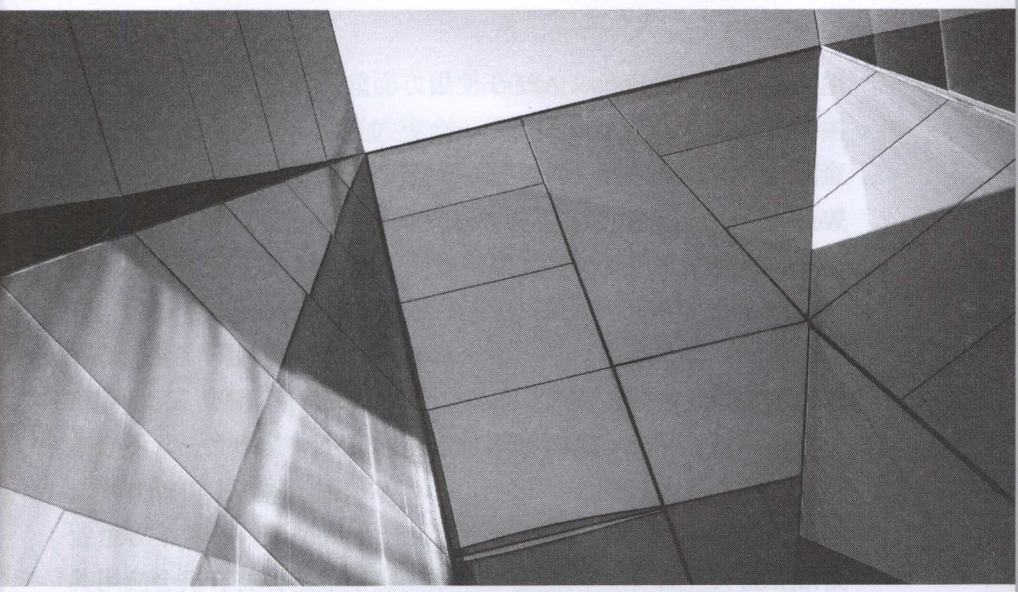
总而言之，此应用使用声明式安全模型完成了如下事情：

- 将对端点的访问限制为仅通过加密连接进行
- 将对 WebSocket 端点的某些形式的访问仅限于某些已经过身份认证的用户
- 允许放开未认证用户对某一端点的访问

使用运行时中的一些 Java WebSocket 安全 API，我们可以查询安全模型，以便根据访问应用的用户来区分 WebSocket 的行动。

7.4　本章小结

本章中学习了如何使用 Web 容器的声明式安全模型将对 Java WebSocket 服务器端点的访问限制为某些用户。我们学习了认证尝试访问 Java WebSocket 服务器端点的用户身份的可用方法，并且也学习了如何确保 Java WebSocket 服务器端点仅可通过加密连接访问。另外，还学习了运行时访问 Java WebSocket 时，开发人员必须加密连接并且具备已认证客户端的身份。最后，介绍了 Stock Account 应用，它把安全模型应用到一个包含有个人数据的经纪人账户上，并且依赖于用户的不同来提供服务级别。



第8章

Java EE 平台中的 WebSocket

Java WebSocket API 是 Java EE 7 平台的一个标准部件，所以对于所有使用支持 Java EE 7 API 的应用服务器的开发人员来说，都可以使用 Java WebSocket API。本书到目前为止，我们专注的应用都只是使用了 Java WebSocket API，以及 Java SE 平台的一些通用类库，除此之外没有使用其他的 API。这使得我们能更详细

地了解 Java WebSocket API 在使用方面的所有重点，但现实世界中很多 WebSocket 端点都属于一个更大企业应用的一部分，这些企业应用会访问各种各样的服务和其他技术。特别是，考虑到 WebSocket 协议的由来(解决 Web 应用的各种轮询形式)，以及在 Web 浏览器中对 WebSocket 协议更高层的支持，所以把 WebSocket 作为一个更大 Web 应用的一部分会是一种常态。

因此，如果不作一些关于 Java EE 和 Web 应用与 WebSocket 协作的介绍，本书会显得不完整。已有很多书专门讲解在 Java EE 平台中如何使用 Java API 和组件，但本书不是其中之一。所以本章会关注如下关键问题：如何让 Java WebSocket 端点与 Web 容器中的其他组件发生联系、如何让 Java WebSocket 端点与企业 JavaBean 发生联系。以这两个联系为出发点，你将有机会应用其他 Java EE 技术来扩展 Java WebSocket 端点的范围和能力。

8.1 Java EE 平台中 Java WebSocket 的角色

从 Java EE 平台架构图(参见图 8-1)中可看出 Java WebSocket API 在 Web 容器中的角色。

Java EE 平台包含了两个服务器端容器：Web 容器和 EJB 容器，这两个容器都驻留和管理开发人员创建的组件。Web 容器驻留了 Java Servlet 及构建于 Java Servlet 模型之上的技术体系，如 JSP(JavaServer Pages)和 JSF(JavaServer Faces)、WebService 端点，当然还有 WebSocket 端点。EJB 容器则驻留了各种各样的企业 JavaBean。这两个服务器容器为开发人员提供了各种服务，包括 JDBC、Java Persistence(从外部数据库读写应用数据)、JMS(JavaMessage Service，用于应用服务器间的可靠消息传递)、

各种安全服务、事务管理器以及 JavaMail。典型的 Java EE 应用持有大量 Web 组件，可能还持有一部分 EJB 组件，通过 Web 组件或者 EJB 组件能直接与数据库交互。用户通过浏览器客户端可访问 Java EE 应用，也可以通过富客户端应用访问，包括 Java 或者 JavaFX 富客户端应用。浏览器交互是发生在加密或者开放的 HTTP 和 WebSocket 连接(这些连接使得 Web 组件和 EJB 组件或者数据层紧密结合)之上的，富客户端交互也是基于这些相同的协议。另外，除了通过 Web 协议与 Web 层通信之外，Java EE 平台还包含了一个管理客户端应用的容器(称为应用客户端容器)，它能直接与 EJB 组件通信。

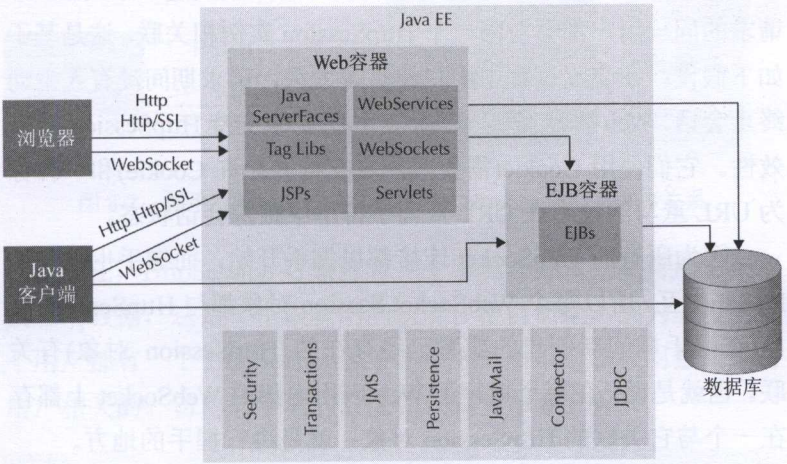


图 8-1　Java EE 平台架构

考虑到 WebSocket 端点在 Web 容器中的位置，很自然地就会想到与同一应用中其他 Web 组件共享应用状态，无论该应用状态是只与某个客户端关联，还是与应用的所有客户端均共享。同样，为了能与 EJB 层的预先存在的组件交互，很自然地就会想从 WebSocket 端点中调用 EJB。

以上两个用例是我们重点关注的：一旦 WebSocket 端点能够与其他 Web 组件共享数据，并且能够与 EJB 组件发生交互，那么它就成了广义 Java EE 应用的一部分。

8.2 共享 Web 应用状态

在 Web 容器中共享应用状态的关键是在于获取如下对象的引用：javax.servlet.ServletContext 和 javax.servlet.http.HttpSession。

HttpSession 对象代表了同一 HTTP 客户端与 Web 应用之间的一系列 HTTP 交互。也就是说，从浏览器客户端发起一系列 HTTP 请求的同一用户都会与同一个 HttpSession 实例相关联。这是基于如下假设：在交互过程中用户没有等太久，请求期间没有人主动终止会话。Web 容器提供了不止一种机制来支持 HttpSession 的有效性。它们使用 Cookie(需要客户端支持并允许 Cookie)和一种称为 URL 重写的技术在 URL 查询字符串中跟踪会话状态。

因为所有的 WebSocket 连接都以握手开始，而握手也是一种 HTTP 交互，所以每个 WebSocket Session 对象都与 HttpSession 对象(在握手期间新创建的或者已经存在的 HttpSession 对象)有关联。也就是说，在每个运行于 Web 应用内部的 WebSocket 上都存在一个与它关联的 HttpSession 对象：就是进行握手的地方。

从更实际方面来讲，从 Web 页面里的 JavaScript WebSocket 端点创建的任何 WebSocket 连接都与同一个 HttpSession 对象(就是页面被加载时创建的 HttpSession 对象)相关联。假设你有一个 Web 应用，它由一个 Web 页面(该页面包含了 3 个 JavaScript WebSocket 客户端端点)和 Java WebSocket 服务端端点组成。假设 JavaScript WebSocket 客户端端点连接 Java WebSocket 服务器端点，然后用

户每次下载 Web 页面引起 WebSocket 连接，就会有一个 HttpSession 对象与 3 个独立的 WebSocket Session 相关联。图 8-2 中显示了当两个用户使用该 Web 应用时的场景。

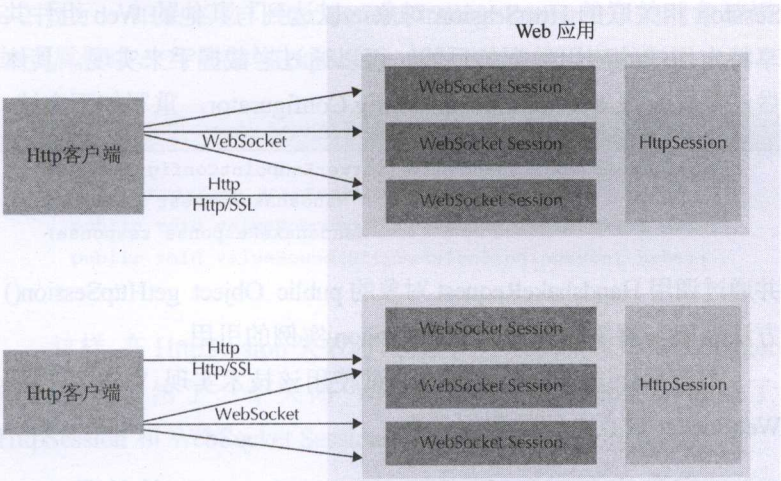


图 8-2　握手时 WebSocket Session 与 HttpSession 之间的关系

现在，HttpSession 对象持有一个字典，开发人员可用它来存放应用数据，这是一个很有用的功能。很自然地，因为应用的每个用户都有一个 HttpSession 实例相对应，所以这些应用数据是与用户相关的。访问该字典的方法如代码清单 8-1 所示。

代码清单 8-1：HttpSession 的用户属性

```
public Object getAttribute(String name)
public Enumeration<String> getAttributeNames()
public void setAttribute(String name, Object value)
```

对于任何能访问 HttpSession 的组件来说，这是一个非常有用的共享对象。所有的 Java Servlet 和基于 Java Servlet 的技术都能

访问 HttpSession 对象，所以它们能够共享某个特定用户的应用状态。

幸运的是，Java WebSocket 端点也能访问与它的 WebSocket Session 相关联的 HttpSession 对象，以达到与其他的 Web 组件共享特定用户的应用数据的目的。可以通过拦截握手来实现，具体做法是自定义 ServletEndpointConfig.Configurator，重写如下方法

```
public void modifyHandshake(ServerEndpointConfig sec,
                            HandshakeRequest request,
                            HandshakeResponse response)
```

并通过调用 HandshakeRequest 对象的 public Object getHttpSession() 方法获取与握手相关联的 HttpSession 实例的引用。

本章后面会有一个示例，通过使用该技术实现 Web 组件和 WebSocket 端点之间的数据共享。

8.2.1 HttpSession 与 WebSocket Session 的关联

一旦应用运行了一段时间，WebSocket Session 实例与 HttpSession 实例之间的关联关系可能变得不再成立。如果 HttpSession 被终止(可能因为显式地失效也可能是超时)，就不能假定 WebSocket 实现会关闭任何与该 HttpSession 关联的 WebSocket Session。只有当用户证实了 Web 应用和 WebSocket 端点被安全约束保护时，WebSocket 实现才会真正关闭与已终止 HttpSession 相关联的 WebSocket Session。你一定不希望受保护的资源在授权状态结束后还继续保持有效。

也就是说如果想在 HttpSession 和 WebSocket Session 之间保持一个更强的关联关系，还需要做些工作。

幸运的是，Servlet API提供了至少一种方法来完成它。

HttpSession属性有个非常有用的特性，正如之前描述的，就是当HttpSession终止(通过显式地失效或超时)时，Web容器会把它(HttpSession属性)从HttpSession中清除。正如下面列出的代码清单8-2 所示，如果实现了HttpSessionBindingListener接口的对象被添加到属性中，当它在与HttpSession解除绑定时会收到通知。

代码清单 8-2：javax.servlet.http.HttpSessionBindingListener

```
public interface HttpSessionBindingListener {
  public void valueUnbound(HttpSessionBindingEvent hsbe);
  public void valueBound(HttpSessionBindingEvent hsbe);
}
```

这样，在 HttpSession 失效时，通过该方法就能通知 WebSocket 端点实例，给了一个关闭 WebSocket 连接的机会，保证了 HttpSession 和 WebSocket Session 之间的强关联性。

8.2.2　HttpSession 示例

下面通过一个 HttpSession 示例来实际地了解上述机制。该示例应用有一个 JavaScript 客户端，它连接了一个 Java WebSocket 服务器端点。其中，Java WebSocket 端点(EndpointWithHttpSession 类)管理了 WebSocket Session 和 HttpSession 之间的强关联性，无论 HttpSession 何时终止，都会关闭 WebSocket 连接。

应用启动后，点击页面上的按钮将会给服务器端点发送一个消息，会按需创建 WebSocket 连接。点击另外一个按钮会终止 HttpSession。从图 8-3 中可看出，当 WebSocket Session 确立时，服务器端点实例就能知道它对应的 HttpSession，当用户使 HttpSession 失效时，会关闭 WebSocket Session。

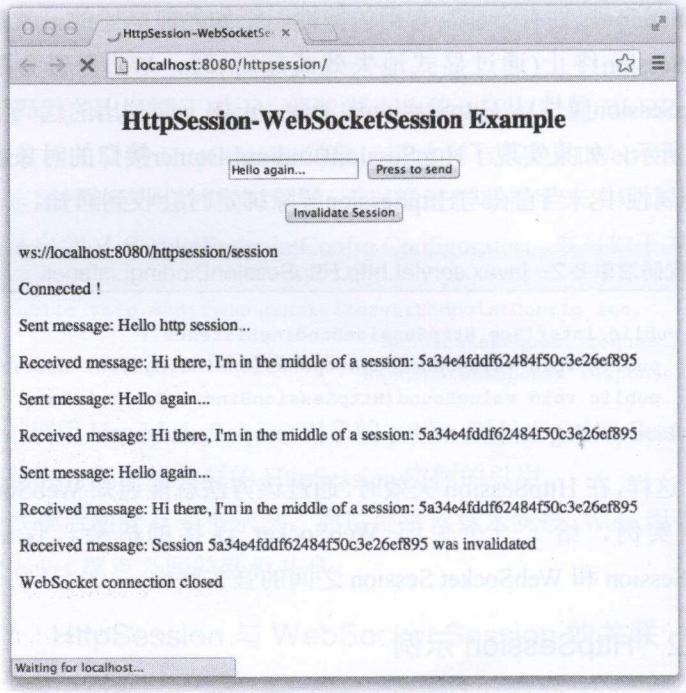


图 8-3　终止会话

WebSocket 端点 EndpointWithHttpSession 通过 HttpSessionConfigurator 获得对 HttpSession 的引用，HttpSessionConfigurator通过拦截握手并调用HandshakeRequest的getHttpSession() 方法获得 HttpSession 对象，并把它放入 ServerEndpointConfig 对象的 user 属性中。我们看看 EndpointWithHttpSession 在它的打开事件处理方法中对 HttpSession做了什么。如代码清单8-3所示。

代码清单 8-3：EndpointWithHttpSession 的@OnOpen 方法

```
@OnOpen
```

```
public void startConnection(Session session,
    EndpointConfig config) {
  this.session = session;
  HttpSession httpSession =
      (HttpSession) config.getUserProperties()
        .get(HttpSessionConfigurator.HTTP_SESSION);
  HttpSessionInvalidationListener l =
       new HttpSessionInvalidationListener(this);
  httpSession.setAttribute("httpsession-listener", l);
  httpSession.setMaxInactiveInterval(5);
}
```

可以看到该方法把WebSocket Session的引用保存为一个实例变量，然后从EndpointConfig对象的用户属性中获得HttpSession (该WebSocket实例所属的HttpSession)的引用。紧接着该方法创建了一个回调对象HttpSessionInvalidationListener，并把它放到了HttpSession的属性中。如代码清单8-4所示是HttpSessionInvalidationListener的代码，当它与HttpSession解绑时会回调EndpointWithSession。

代码清单 8-4：HttpSessionInvalidationListener

```
public class HttpSessionInvalidationListener
       implements HttpSessionBindingListener {
  private EndpointWithHttpSession ewhs;

  public HttpSessionInvalidationListener(
      EndpointWithHttpSession es) {
    this.ewhs = es;
  }

  public void valueUnbound(HttpSessionBindingEvent hsbe) {
    ewhs.notifySessionUnboundMe(hsbe.getSession());
  }

  public void valueBound(HttpSessionBindingEvent hsbe) {}
}
```

因为该应用从没显式地调用 HttpSession 的 removeAttribute() 方法来移除这个属性，所以唯一能够从 HttpSession 中移除它的时机就是在 HttpSession 被终止时。

该应用包含了一个简单的名为 InvalidateServlet 的 Java Servlet，用来显式地使 HttpSession 失效，在一个合适的延时之后，会引起 Web 页面的重新加载并清除输出。代码清单 8-5 是该类的源码：

代码清单 8-5：InvalidateServlet 的处理方法

```
protected void processRequest(HttpServletRequest request,
    HttpServletResponse response)
      throws ServletException, IOException {
  request.getSession().invalidate();
  try {
   Thread.sleep(2000);
  } catch (Exception r) {}

 response.sendRedirect("index.jsp");
 }
```

这相应地会调用 HttpSessionInvalidationListener，它又会调用 EndpointWithHttpSession 端点实例的 notifySessionUnboundMe()方法。如代码清单 8-6 所示。

代码清单 8-6：EndpointWithHttpSession 的通知处理方法

```
public void notifySessionUnboundMe(HttpSession s) {
 try {
  this.session.getBasicRemote().sendText("Session " +
     s.getId() + " was invalidated");
  this.session.close(
   new CloseReason(CloseReason.CloseCodes.NORMAL_CLOSURE,
      "HttpSession ended"));
 } catch (Exception r) {
```

```
            r.printStackTrace();
        }
    }
```

正如代码中所见到的，在关闭 WebSocket 连接并完成 WebSocket连接与HttpSession之间的强关联性之前，相应地会给浏览器发送一个消息。

如果该示例应用运行得更久一些，就会注意到如果 HttpSession 超时，WebScoket Session 同样会关闭，在 Web 页面上也能看到相同的消息。HttpSession 在超时间隔之后是容许超时的，但是 Web 容器中的 HttpSession 对象可能并不会在超时间隔达到时立即失效。然而，如果想看看超时效果，在 HttpSession 超时后，可以重新加载 Web 页面以创建一个新的 HttpSession，这将会使老的 HttpSession 失效并关闭 WebSocket 连接。

既然我们已经知道如何共享某个特定用户的应用状态，那么使所有用户共享应用状态也是一件非常简单的任务，使用 ServletContext 就可以轻松地完成。通过使用 HttpSession 对象的 public ServletContext getServletContext()方法能获得 ServletContext 的引用。因此，举个示例，在 WebSocket 端点中，如果想与同一 Web 应用中的其他 Web 组件(当然也包括其他的 WebSocket)共享应用状态，只需要 ServletContext 即可。在 WebSocket 端点的 EndpointConfig 中，通过使用自定义配置器能获得 ServletContext，如代码清单 8-7 所示。

代码清单 8-7：使用配置器获得 ServletContext

```
public class ServletContextConfigurator
        extends ServerEndpointConfig.Configurator {
    public static String SERVLET_CONTEXT = "servlet-context";
```

```
    public void modifyHandshake(ServerEndpointConfig sec,
        HandshakeRequest request,
        HandshakeResponse response) {
      if (sec.getUserProperties().get(SERVLET_CONTEXT) != null) {
        ServletContext servletContext =
        ((HttpSession) request.getHttpSession())
          .getServletContext();
        sec.getUserProperties().put(SERVLET_CONTEXT,
          servletContext);
      }
    }
}
```

8.3 WebSocket 端点使用 EJB

在 Java EE 平台中运行时，WebSocket 服务器端点是上下文无关 Bean。这就是说端点实例的创建和销毁是被 Java EE 平台管理的，也允许 Java EE 平台注入其他的 Java EE 组件到端点实例中。这些组件包括企业 JavaBean、数据库连接，以及 Java EE 平台的 JNDI 名称空间内的其他资源。有很多介绍 Java EE 平台中依赖注入的书，所以这里我们只是简单地了解一些最有用的组件：企业 JavaBean。

为了能把 EJB 注入 WebSocket 端点中，首先需要在 Web 应用中允许依赖机制。可以通过在 Web 应用的 WEB-INF 目录中增加一个默认的 beans.xml 来实现，beans.xml 文件的内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
</beans>
```

一旦允许依赖注入，就可以使用@EJB 注解把 EJB 注入端点中。代码清单 8-8 演示了在端点中注入 EJB。

代码清单 8-8：简单的 EJB 注入

```
import javax.ejb.EJB;

@ServerEndpoint("/my-endpoint")
public class MyEndpoint {
   @EJB
   private MyEJB myEJB;
...
}
```

因为当新的客户端成功完成握手时，是由 Java EE 平台负责创建 MyEndpoint 类实例的，所以就能够给 MyEndpoint 类实例注入一个合适的 MyEJB 企业 JavaBean 实例。这就是说等到 MyEndpoint 类的打开处理方法被调用时，myEJB 实例变量已经初始化完成并等待被调用了。

注入 WebSocket 端点的 EJB 分为两种类型：有状态的会话 Bean 和单例 Bean。在前面的代码中，如果 MyEJB 组件是一个有状态的会话 Bean，每当因为新连接而创建新 WebSocket 端点时，Java EE 平台都会注入一个新的 MyEJB 组件实例到 WebSocket 端点中。这使得这种 Bean 在绑定特定应用状态与 WebSocket 连接时非常有用。如果 MyEJB 组件是一个单例 Bean，每当 WebSocket 端点被实例化时，Java EE 平台都会注入相同的 MyEJB 组件实例到 WebSocket 端点实例中。这使得注入单例 EJB 到 WebSocket 端点中对于共享所有连接公用的应用状态来说是一个非常有用的方法。

关于如何在应用中引入 EJB，我们已经简单地介绍了两个重

要的使用场景，接下来看一个简单的示例。

EJB 示例

在如代码清单 8-9 所示的 EJB 示例中，我们有个单一服务器端点，名为 EndpointWithEJBs。

代码清单 8-9：EndpointWithEJBs

```
import javax.ejb.EJB;
import javax.websocket.OnError;
import javax.websocket.OnMessage;
import javax.websocket.server.ServerEndpoint;


@ServerEndpoint("/endpointwithejbs")
public class EndpointWithEJBs {
  @EJB
  private MyStatefulEJB myEJB;
  @EJB
  private MySingletonEJB mySingleton;

  @OnMessage
  public String hiThere(String message) {
    return "Hi. I have two EJBs: <br> " + myEJB.getMessage() +
        "<br> " + mySingleton.getMessage();
  }

  @OnError
  public void error(Throwable t) {
    System.out.println("Error: " + t.getMessage());
  }
}
```

你会发现这只不过是第 1 章的 Echo 示例的一个简单演化，除了响应任何传入的消息之外，该服务器端点有两个 EJB 用来生成

消息：一个有状态的会话 Bean 和一个单例 Bean。通过使用@EJB 注解，Java EE 平台会负责确保这些指向 EJB 的实例变量在打开处理方法被调用(当有新的客户端连接到 EndpointWithEJBs 端点时)之前就正确地完成初始化。每个 EJB 做如下相同的事情：当第一次被创建时，会标记当前时间。无论哪个 EJB 的 getMessage() 方法被调用，都会返回 EJB 被创建时刻的时间字符串。通过创建时间来区分同一 Bean 的不同实例是一个非常粗略但又有效的方法。

当第一个客户端连接到 EndpointWithEJBs 时，在 Web 页面上点击按钮，会看到如图 8-4 所示的输出内容，直接显示了 WebSocket 端点引用的 EJB 实例是被同时创建的。然而，如果重新加载页面，从而导致一个到 WebSocket 端点的新的 WebSocket 连接，当点击按钮时会看到有些区别，如图 8-5 所示。

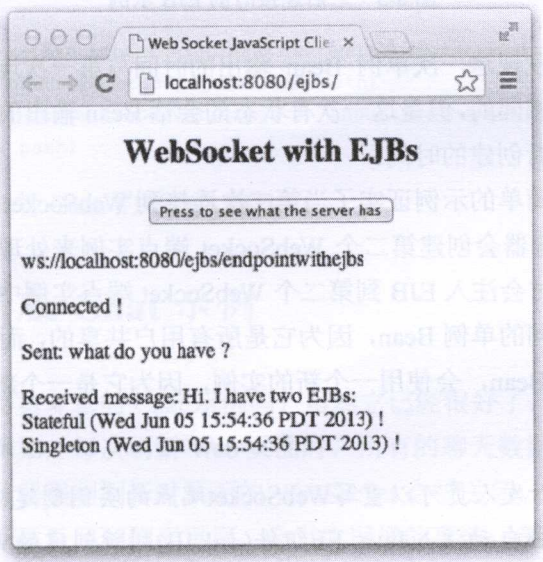


图 8-4　第一次连接时的 EJB 示例

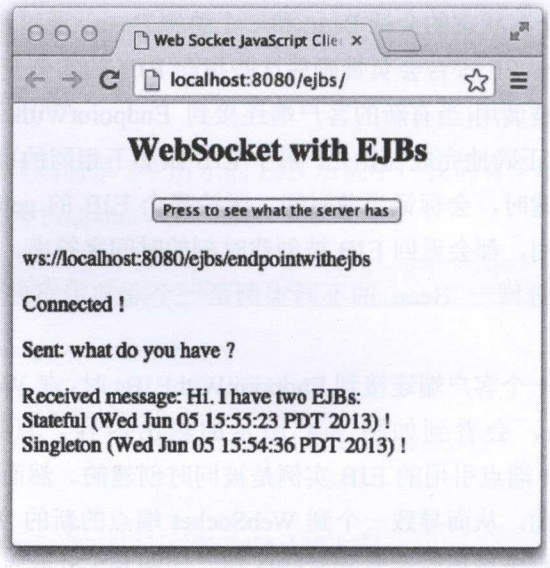


图 8-5 之后连接时的 EJB 示例

可以发现这一次单例 Bean 输出的时间与第一次连接时的输出时间是相同的，但是这一次有状态的会话 Bean 输出的时间是不同的，是新创建的时间。

这个简单的示例证实了当第二次连接到 WebSocket 端点时，Java EE 容器会创建第二个 WebSocket 端点实例来处理新的连接请求，同时会注入 EJB 到第二个 WebSocket 端点实例中。注入时会使用相同的单例 Bean，因为它是所有用户共享的，而对于有状态的会话 Bean，会使用一个新的实例，因为它是一个新的连接。

注意：

高级开发人员可以重写WebSocket端点的实例创建机制，而这样做会失去自动注入Java EE组件(如EJB)到新创建的WebSocket端点中的能力。

在第4章中了解到，通过创建一个自定义的ServerEndpointConfig.Configurator类并且重写public T getEndpointInstance(Class<T> c)方法，能重写服务器端点的实例创建机制。如果没有重写该方法，Java EE平台会利用已托管Bean的可注入特性为你创建所有的实例。

通过自己完成实例化，以及使用如下任一WebSocketContainer的方法(取决于WebSocket端点的类型)，可以管理自己的客户端端点实例。

```
Session connectToServer(Endpoint endpointInstance,
    ClientEndpointConfig cec, URI path)
Session connectToServer(Object annotatedEndpointInstance,
    URI path)
```

如果你坚持使用重载的 connectToServer()方法(用端点类取代实例)：

```
Session connectToServer(Class<? extends Endpoint> endpointClass,
   ClientEndpointConfig cec,URI path)
Session connectToServer(Class<?> annotatedEndpointClass,
   URI path)
```

就会获得托管 Bean 的可注入特性。

8.4　新版 Chat 示例

还记得第 4 章的 Chat 示例吗？虽然它已经很好了，但还是存在局限。例如，每次刷新 Web 页面时，所有的聊天数据会丢失，因为当浏览器强制刷新时页面的 WebSocket 会被关闭。如果加载了其他页面再返回到聊天页面，会发生同样的事情。在新版 Chat 应用中，我们将应用一些本章学过的简单技术在 HttpSession 中存

放聊天信息，同时我们将存放群体聊天信息的聊天记录代码转移到 EJB 中，使得这些信息能被多个 Java Servlet 共享。

在看代码之前，看看应用在用户角度方面有什么改变。

第一次登录进来时，会在主页面上看到一些新的元素，如图 8-6 所示。注意页面的底部，HTML 代码中显示了用户名，并有一个 Find out more 链接，点击该链接会进入聊天数据页面，如图 8-7 所示。再次注意前一页面的 HTML 代码中已经记录了你的用户名。同时，当你回到主页面再以另外一个身份登录时，会产生一条消息说你离开房间一会儿(见图 8-8)。当你返回到聊天主页面时，会发现还能看到所有的聊天数据，此外还会告知所有人你回来了(见图 8-9)。

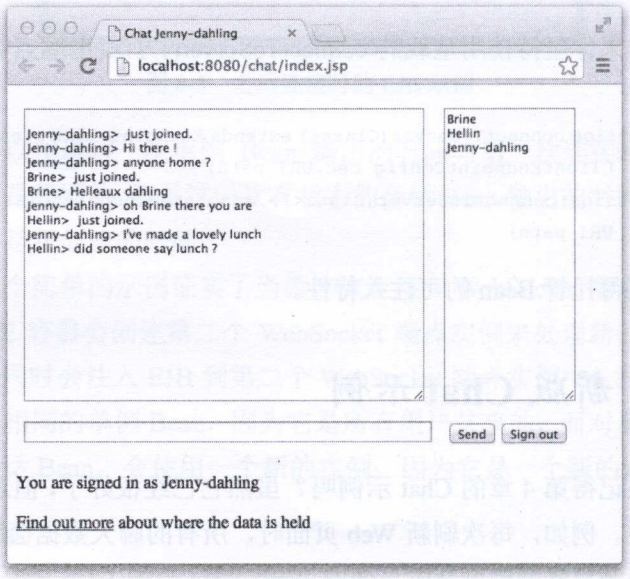


图 8-6 聊天主窗口

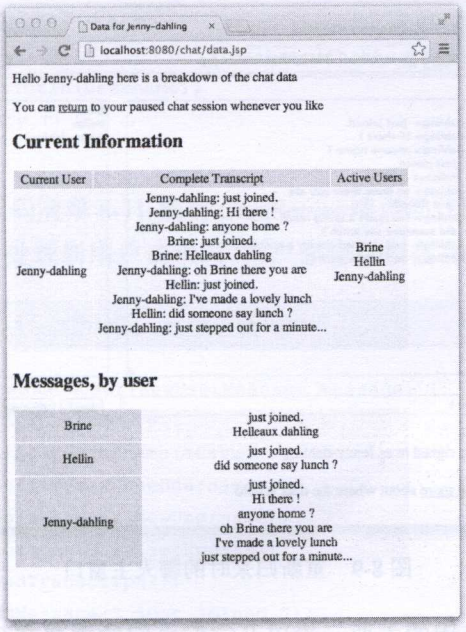


图 8-7　聊天数据窗口

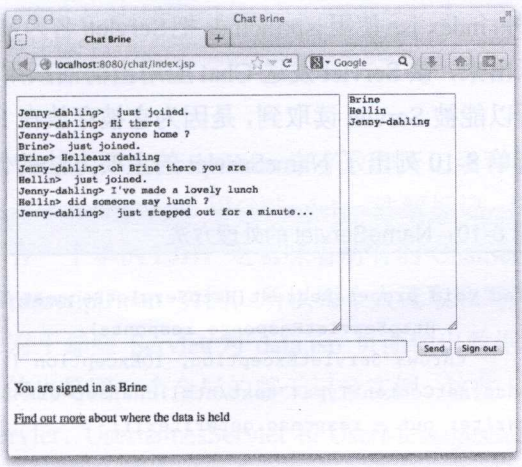


图 8-8　在你离开时的聊天主窗口

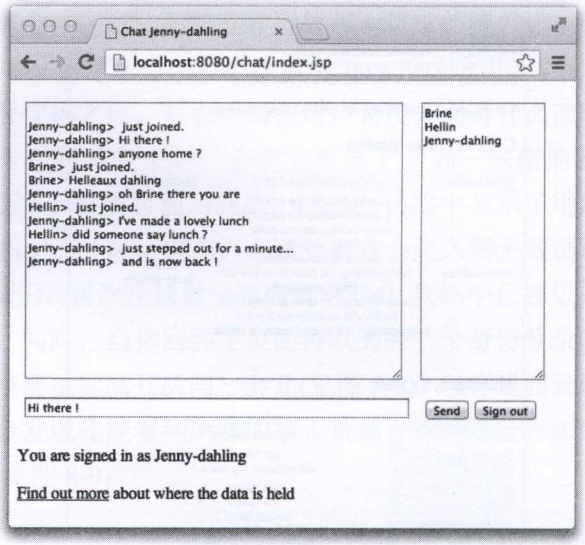


图 8-9 重新归来时的聊天主窗口

先看看应用的主要元素以及它们之间的主要交互情况。聊天交谈是在 index.jsp 页面的 JavaScript 客户端与 ChatServer 端点之间完成。此外，index.jsp 使用 jsp:include 把 Servlet(名为 NameServlet)的输出显示出来，该 Servlet 发送 Chat 应用当前活跃用户的名字。用户名之所以能被 Servlet 读取到，是因为它被存放在 HttpSession 中。代码清单 8-10 列出了 NameServlet 的主要方法的代码：

代码清单 8-10：NameServlet 的处理方法

```
protected void processRequest(HttpServletRequest request,
                HttpServletResponse response)
            throws ServletException, IOException {
  response.setContentType("text/html;charset=UTF-8");
  PrintWriter out = response.getWriter();
  try {
    HttpSession session = request.getSession();
```

```
        String username =
          (String) session.getAttribute(ChatServer.USERNAME_KEY);
        out.println(username);
      } finally {}
    }
```

正如从代码清单 8-11(摘自 ChatServer 端点的代码)中看到的，当 ChatServer 处理新用户请求时，用户名会存放在 HttpSession 中。

代码清单 8-11：ChatServer 处理新用户

```
void processNewUser(NewUserMessage message) {
  String newUsername =
  this.validateUsername(message.getUsername());
  this.confirmUser(newUsername);
  this.registerUser(newUsername);
  this.broadcastUserListUpdate();
  this.sendTranscript();
  this.addMessage(" just joined.");
}
...
private void registerUser(String username) {
  this.httpSession.setAttribute(USERNAME_KEY, username);
  this.updateUserList();
}
```

ChatServer 使用 TranscriptBean 来存放聊天记录和保持当前活跃用户列表。TranscriptBean 用@Singleton 注解标记，因此对每个应用来说都有一个单例 EJB。这意味着所有的 ChatServer 实例共享同一个 TranscriptBean 实例，所以每个人读取和写入的都是同一个记录。对于那些 Servlet(为 data.jsp 页面输出数据的 Servlet)来说，使用的也是同一个全局的聊天记录实例。这些 Servlet 包括 TranscriptServlet、UsernamesServlet 和 UserMessagesServlet，它们都使用如下声明并依赖 Java EE 平台正确地初始化引用：

```
@javax.ejb.EJB
private TranscriptBean transcriptBean;
```

ChatServer 使用了之前学过的配置器技术 ChatServerConfigurator 来获得 HttpSession 引用并把它存放在 EndpointConfig 中。

从图 8-10 中可以清晰地看到 Chat 应用的主要组件，这些组件存放了应用的关键信息。

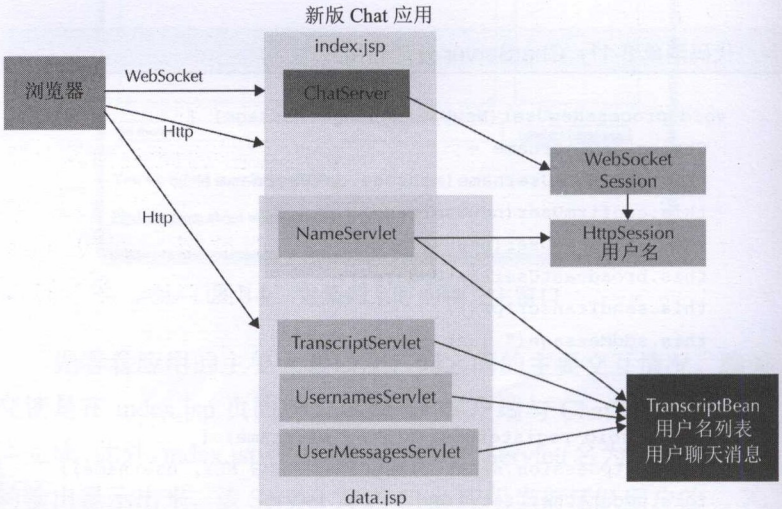


图 8-10　新版 Chat 应用体系结构

最后给读者留个作业来更详细地探索代码，看看究竟是在哪里建立联系的，同时也留个练习：当 HttpSession 超时发生时关闭聊天会话。

8.5　本章小结

在本章中，我们最开始探讨了如何扩展 WebSocket 服务器端

点的范围，方法是在 Java EE 平台中与其他 Java EE 组件进行集成。我们学习了当前 WebSocket 端点是如何关联 HttpSession 的以及如何定位 Web 容器的 ServletContext，探讨了与 Web 容器中其他 Web 组件共享用户状态和全局应用状态的可能性。我们还探讨了如何在 WebSocket 服务器端点中注入其他的 Java EE 组件(如 EJB)，以便使用原有的中间件 EJB 组件，以及利用 EJB 组件已经提供的服务(如事务管理和数据库连接等)。

最后，我们把第 4 章的 Chat 应用进行了改进，使得它能感知 HttpSession，通过单例 EJB 和 Web 组件(提供了聊天室的当前状态报告)来共享它的状态，以便当用户浏览其他页面再返回聊天页面时能够恢复聊天会话。

我们希望能激发读者更深入探索的欲望，利用 Java WebSocket 和其他 Web 及 Java EE 组件来搭建时尚的、迷人的和强大的 Web 应用！

Oracle 官方系列丛书

Oracle数据库

Java WebSocket编程 开发、部署和保护动态Web应用
精通lambda表达式：Java多核编程
Oracle Database 12*c* PL/SQL开发指南（第7版）
OCA认证考试指南(1Z0-062)：Oracle Database 12*c* 安装与管理
OCA认证考试指南(1Z0-061)：Oracle Database 12*c* SQL 基础
Oracle Database 12*c*完全参考手册（第7版）
Oracle NoSQL数据库：实时大数据管理
Oracle大数据解决方案
专业级Oracle Database 12*c*安装、配置与维护
云构建与管理——使用Oracle Enterprise Manager 12*c*
深入解析Oracle Enterprise Manager Cloud Control 12*c*
精通Oracle Database 12*c* SQL & PL/SQL编程(第3版)

Java编程

Java 8编程参考官方教程(第9版)
Java 8编程入门官方教程(第6版)
Java EE 7 & HTML5应用开发——构建和部署同时支持桌面和移动设备的动态、高性能企业级应用

MySQL数据库

MySQL Workbench数据建模与开发
Effective MySQL之SQL语句最优化
Effective MySQL之备份与恢复
Effective MySQL之深入解析复制技术
Oracle Database 11*g* & MySQL 5.6开发手册

清华大学出版社
http://www.tup.com.cn

精通Java WebSocket应用开发

充分利用最先进的通信技术构建动态企业级Web应用。《Java WebSocket编程 开发、部署和保护动态Web应用》一书由Java WebSocket编程权威专家撰写，提供了实际的开发策略和详尽的示例应用。本书诠释了如何设计客户端/服务器应用、与全双工消息通信协作、建立连接、创建端点、处理路径映射以及保护数据。你也将学到如何加密Web传输并且使用Java WebSocket增强既有应用。

主要内容

- 使用Java WebSocket API开发Web应用
- 创建和发布注解式端点和编程式端点
- 管理WebSocket端点的生命周期事件
- 维护端点生命周期中可靠的连接
- 管理同步和异步消息通信
- 为复杂消息通信定义编码和解码策略
- 配置消息通信超时、大小限制和异常
- 消息路径映射及将入站URI路由到Web容器
- 保护数据、认证用户以及加密连接

作者简介

Danny Coward是Oracle的首席架构师和Web架构师。他是Java EE、Java SE/JavaFX中WebSocket Java API的规范领导者。他在Oracle主导WebSocket工作，他是Java WebSocket编程方面的权威专家。

Oracle Press™

清华大学出版社数字出版网站
WQBook 书文局泉
www.wqbook.com

McGraw-Hill
全球智慧中文化
http://www.mheducation.com

ISBN 978-7-302-40807-9
9 787302 408079 >
定价：39.00元